Protection Unit for Radiation Induced Errors in Flash Memory Systems

by

Bevan Bryer

Thesis presented in partial fulfillment of the requirements for the degree of Masters of Science in Engineering at the University of Stellenbosch

December 2004

Supervisor: Prof. P.J. Bakkes

Declaration

I declare that the contents of this thesis is original and my own work unless otherwise stated and that it has not, to my knowledge, been published in any part or as a whole at any other university in order to obtain a degree.

Synopsis

Flash memory and the errors induced in it by radiation were studied. A test board was then designed and developed as well as a radiation test program. The system was irradiated. This gave successful results, which confirmed aspects of the study and gave valuable insight into flash memory behaviour. To date, the board is still being used to test various flash devices for radiation-harsh environments.

A memory protection unit (MPU) was conceptually designed and developed to monitor flash devices, increasing their reliability in radiation-harsh environments. This unit was designed for intended use onboard a micro-satellite. The chosen flash device for this study was the K9F1208XOA model from SAMSUNG. The MPU was designed to detect, maintain, mitigate and report radiation induced errors in this flash device. Most of the design was implemented in field programmable gate arrays and was realised using VHDL.

Simulations were performed to verify the functionality of the design subsystems. These simulations showed that the various emulated errors were handled successfully by the MPU.

A modular design methodology was followed, therefore allowing the chosen flash device to be replaced with any flash device, following a small reconfiguration. This also allows parts of the system to be duplicated to protect more than one device.

Opsomming

'n Studie is gemaak van "Flash" geheue en die foute daarop wat deur radiasie veroorsaak word. 'n Toetsbord is ontwerp en ontwikkel asook 'n radiasie toetsprogram waarna die stelsel bestraal is. Die resultate was suksesvol en het aspekte van die studie bevestig en belangrike insig gegee ten opsigte van "flash" komponente in radiasie intensiewe omgewings.

'n Geheue Beskermings Eenheid (GBE) is konseptueel ontwerp en ontwikkel om die "flash" komponente te monitor. Dit verhoog die betroubaarheid in radiasie intensiewe omgewings. Die eenheid was ontwerp met die oog om dit aan boord 'n mikro-satelliet te gebruik. Die gekose "flash" komponent vir die studie was die K9F1208X0A model van SAMSUNG. Die GBE is ontwerp om foute wat deur radiasie geïnduseer word in die "flash" komponent te identifiseer, herstel en reg te maak. Die grootste deel van die implementasie is gedoen in "field programmable gate arrays" and is gerealiseer deur gebruik te maak van VHDL.

Simulasies is gedoen om die funksionaliteit van die ontwikkelde substelsels te verifieer. Hierdie simulasies het getoon dat die verskeie geëmuleerde foute suksesvol deur die GBE hanteer is.

'n Modulre ontwerpsmetodologie is gevolg sodat die gekose "flash" komponent deur enige ander flash komponent vervang kan word na gelang van 'n eenvoudige herkonfigurasie. Dit stel ook dele van die sisteem in staat om gedupliseer te word om sodoende meer as een komponent te beskerm.

Acknowledgements

The author would like to thanks the following people:

- Prof. P.J. Bakkes (thesis supervisor) for his advice
- Carine Loubser for her support and help in finishing this document

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

A	Ampere, SI-unit for electric current
A2D	Analog to Digital
ASCII	American Standard Code for Information Exchange
ASIC	Application Specific IC
CAN	Controller Area Network
CCD	Charge Coupled Device
CHE	Channel Hot Electron
COMM	Communication
ECC	Error Correction Code
EDAC	Error Detection and Correction
EEPROM	Electrically Erasable Programmable ROM
eV	Electron Volt
F-N	Fowler-Nordheim
FET	Field Effect Transistor
FIFO	First-in First-out
FPGA	Field Programmable Gate Array
G	Giga = 10^9
HDL	Hardware Descriptive Language
Hz	Hertz = per second
IC	Integrated Circuit
J	Joule-SI unit for energy
k	kilo =10^3
LEO	Low Earth Orbit
LET	Linear Energy Transfer
M	Mega = 10^6
MMU	Mass Memory Unit
MOSFET	Metal Oxide Semiconductor FET
MPU	Memory Protection Unit
OBC	Onboard Computer

PC	Personal Computer
PCB	Printed Circuit Board
RAD	SI-unit for radiation
RAM	Random Access Memory
ROM	Read Only Memory
RS	Reed Solomon
SAA	South Atlantic Anomaly
SEFI	Single Event Functional Interrupt
SEL	Single Event Latchup
SEU	Single Event Upset
SRAM	Static RAM
TID	Total Ionizing Dose
UART	Universal Asynchronous Receiver Transmitter
V	Volt
VHDL	VHSIC HDL
VHSIC	Very High Speed IC

Chapter 1

Introduction

Designing and building low earth orbit satellites has been an attraction at Stellenbosch University since 1992. A privileged group of engineers got the opportunity in designing and building SUNSAT I (Stellenbosch University Satellite) which was launched in 1999. Following the success of the first satellite engineers at Stellenbosch have been involved with designing a new improved satellite SUNSAT II. The main purpose of the satellite project is to train engineers using practical experience.

This chapter starts off by describing the need for increasing the reliability of the flash-based memory module. The reader is also made aware of the boundaries that exist for the thesis in defining the scope of the project.

1.1 The Need

SUNSAT II is a micro-satellite being developed by post-graduate engineers at Stellenbosch University. The primary use of SUNSAT II is the same as that of SUNSAT I which is imaging.

The images are acquired by the Pushbroom Imager which scans the earth surface creating large amounts of data which has to either be sent directly to an earth station or stored internally in the satellite until required. To store such a large amount of data requires a fairly large memory module or RAMDISK.

The RAMDISK must be designed and built to meet strict requirements of the satellite specifications. These include the RAMDISK's power consumption, size, weight, speed and rigidness. In improving the RAMDISK used in SUNSAT I a smaller, lighter, faster RAMDISK was required to be designed. The main factor contributing to the size and weight of SUNSAT I's RAMDISK was the large physical size and number of its memory

chips. The low density of the SRAM (static random access memory) memory modules resulted in a large number of chips being used to satisfy the memory requirement. Therefore a memory chip with higher density and smaller size was required to improve upon the older system. This lead to the replacement of SRAM with FLASH RAM.

Flash ram is a newer, denser type of electronic memory. Electronic meaning no moving parts. Due to its high density, small size and low power requirements, flash memory has become the memory type of choice for many new technologies, such as digital cameras, cellular phones and even field-programmable gate arrays (FPGA). Flash memory chips come in packages with physical dimensions as small as $20mm \times 12mm$ and can have densities up to 4 Gbit (500 Mbytes). This is extremely dense compared with SUNSAT I's RAMDISK which was 64Mbytes in total!

However, the high density and complex control circuitry within a flash ram chip makes it extremely vulnerable to radiation induced errors. The radiation environment that low-earth-orbit satellites are exposed to is much harsher than that on earth due to the earth being shielded by the magnetosphere. Therefore flash memory systems will be highly affected by the radiation and can be damaged and even become defective if not protected. A completely damaged memory system can render a satellite, such as SUNSAT, virtually non-operational due to the fact that images will not be able to be stored onboard the satellite which will limit image capturing locations to areas in direct view of the ground station. Therefore measures must be taken to protect the memory system from radiation induced errors and increase its reliability in space environment operations.

The object of this thesis therefore, was to determine what effects radiation has on flash memory and develop a system which determines, mitigates and reports the errors that are caused by those effects.

1.2 The Rest of the Document

Chapter 2 investigates the radiation environment and its effects on semiconductors. This was done to give an understanding of what radiation stresses the memory module has to withstand.

In chapter 3 the important aspects of flash memory was investigated. These aspects include the basic understanding of what flash memory is, how it works, the internal structure as well as how flash memory reacts to radiation. A radiation test was also performed which included hardware and software design of a test circuit board. This chapter gives

an understanding of the vulnerabilities of flash memory to radiation and what issues need to be protected in order to reduce those vulnerabilities.

Chapter 4 performs a tradeoff analysis deciding which type of error correcting code to implement and how it should be implemented. As a result, it was decided to implement a Reed-Solomon coding scheme in hardware, using a field programmable gate array.

The concept design for the memory protection unit was performed in chapter 5. This is where all the requirements and subsystems for the main system were discussed and conceptual designs formulated.

Chapter 6 details the low-level design of the MPU. This involves realising the concept designs in hardware and then verifying them by simulations.

Lastly chapter 7 includes the conclusions and recommendations from this study.

Chapter 2

Radiation

This chapter investigates the typical environment to which LEO memory system will be exposed to and the damaging effects that radiation in this environment can have on the main components of that system.

2.1 Basics

2.1.1 Definition

Radiation is the emission of energy as moving particles or as electromagnetic waves. Radiation is comprised of high-energy particles and photons [8]. Semiconductors which are exposed to radiation can be severely damaged.

2.1.2 Types of radiation

Gamma rays and X-rays: These two forms of radiation interact with matter identically. They are short-wavelength forms of electromagnetic or photon radiation. Their reaction with matter is lightly ionizing and highly penetrating [8] and leave no activity in the radiated material. There are two ways in which they originate:
Firstly, when electrons fall into vacancies in the n = 1 or n = 2 levels of an atom, photons are emitted [6]. These photons carry an energy equal to the difference in energies between the two levels which act as the start and end point for the electron that falls into that vacancy. These emitted photons or X-rays are called 'characteristic x-rays'.
Secondly, x-rays are also emitted from a target when it is bombarded by electrons and is referred to as 'bremsstrahlung'. The deceleration of moving charge causes this radiation. Unlike characteristic x-rays, bremsstrahlung has a continuous range of energies due to the fact that deceleration can occur in an infinite number of ways.
X-radiation is an ionizing radiation, see 2.3.2, since charged particles are produced, and react with atoms in three ways:

1. The photo-electric effect, where a photon strikes an electron and loses all its energy to it and disappears.
2. The Compton effect, where a photon collides with an electron and is scattered.
3. Pair Production, where a photon, which is in close proximity to a nucleus, disappears and is replaced by an electron and a positron.

These effects can increase the conductivity of the material through their creation of excess carriers but in the LEO satellite environment the intensity of x-rays are fairly low and therefore considered only a minor importance to this study.

Alpha particles: They are the nuclei of helium atoms, therefore comprised of 2 protons and 2 neutrons giving a mass of 4 and positive charge of 2 units [8]. Alpha particles are produced as a by product when an unstable heavy atom nuclei decays. These particles have high energy and are heavily ionizing but have low penetration, (typically 5MeV energy with a range of 23um in silicon). Due to the low penetration ability of alpha particles, they are of little concern to LEO satellites.

Beta particles, electrons, positrons: Beta particles have the same mass as an electron but have either negative or positive charge. They are created inside the nucleus when a proton becomes a neutron. Because they have very small size and charge they easily penetrate matter, but are easily deflected and are lightly ionizing due to their high velocities.

Neutrons: Neutrons have the same mass as a proton but no charge and are therefore hard to stop. They originate from nuclear reactions and radioactive decay. Due to their lack of charge, they are the primary form of non-ionizing radiation (atomic displacement , see 2.1.3). Stopping a neutron results in the emission of a gamma ray. Neutrons can be classified as either thermal, intermediate or fast, depending on their energy.
Neutron radiation can change the minority carrier densities in bipolar transistors by decreasing minority carrier lifetimes [6]. This degrades the current gain of the transistor and therefore must be taken into account in LEO design.

Protons: The proton is the nucleus of a hydrogen atom and carries a charge of 1 unit. They originate from radioactive decay and nuclear reactions. It is heavy compared to electrons and is therefore harder to deflect. The typical penetration range is tens of micrometers in aluminium at energies in the MeV range.

2.1.3 Units of radiation

Energy: The **joule(J)** is the SI unit of energy but in radiation technology the electron volt(eV) is more frequently used. 1 eV is the amount of energy gained by 1 electron accelerating through the potential difference of 1 volt. $1eV = 1.6 \times 10^{-19}J$. Throughout this thesis, energy will mainly be quoted as MeV (10^6) or keV (10^3).

Flux: This is the number of particles flowing through a defined area per unit time [8]. The unit of flux is $cm^{-2}s^{-1}$. The symbol for the type of particle is normally added.

Fluence: This is the time-integrated flux. Unit is cm^2.

Energy Deposition: The **RAD** is a universal unit of energy deposition. A RAD has been absorbed by a sample when 100 ergs per gram has been deposited [8]. An erg being 10^{-7} joules.

2.2 Radiation environments

There are many environments that can have a degrading effect on silicon devices such as nuclear reactors, space, processing, weapons and controlled fusion. However the space environment is the one which affects this project the most and will therefore be discussed. The particles which comprise the space radiation environment are either trapped by the earth's magnetic belt or passing through the solar system. The main elements are the radiation belts, cosmic rays, solar flares and solar wind.

2.2.1 Radiation Belts

A broad spectrum of energetic charged particles are trapped in the Earth's magnetic field. They form what is called the radiation belts or the Van Allen belts.

Particles that approach the radiation belts are either trapped or deflected. The trapped particles oscillate north-south along the field lines of the magnetosphere. This occurs because the converging field lines at the magnetic poles act as a magnetic mirror, reflecting the particles. There are two main radiation belts, the electron and the proton radiation belts.

The electron belt consists of two zones, the inner and outer zones, where two flux maxima occur (see figure 2.1). In the electron belt, particles have energies up to 7 MeV with the most energetic particles occurring in the 'outer zone'. The inner zone extends to about

2.4 earth radii (1 Earth radii being 6371km) and the outer from 2.8 to 12 earth radii [8]. The contours of the outer zone extend toward the earth in 'horns' of high flux known as 'the polar horns'.

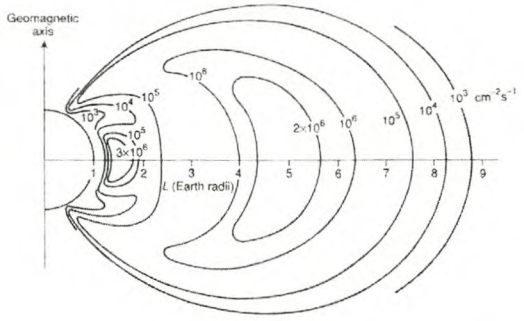


Figure 2.1: *Trapped-electron radiation belts,plotted contours of equal electron flux of energy above 1 MeV ([8] Fig. 2.2)*

The proton belt is simpler in shape than the electron belt (see figure 2.2). Its flux decreases with distance with the more energetic particles found at lower altitudes. The proton belt extends to about 3.8 Earth radii.

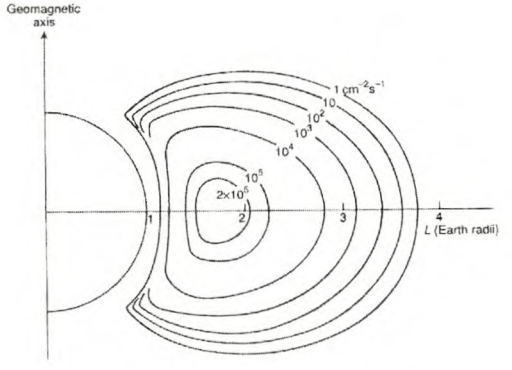


Figure 2.2: *Trapped-proton radiation belts, plotted contours of equal proton flux of energy above 10 MeV ([8] Fig. 2.3)*

An anomaly in the South Atlantic region causes the Earth's trapped radiation belt to dip, increasing fluxes in that region a hundred fold (see figure 2.3). Therefore satellites in LEO will experience large increases in radiation when passing through this region. It is known as the South Atlantic Anomaly (SAA).

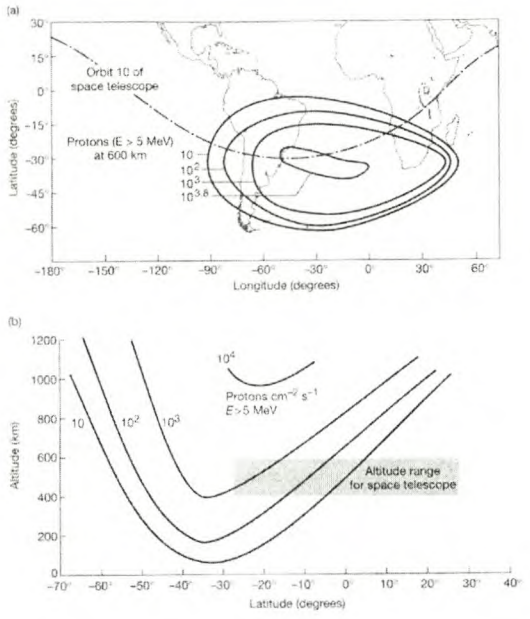


Figure 2.3: *The South Atlantic Anomaly.(a)Flux of protons as a function of latitude and longitude,at an altitude of 600km.The orbit track of the Hubble Space Telescope is shown.(b)Flux of protons as a function of altitude and latitude at a longitude of 35° W ([8] Fig. 2.5)*

2.2.2 Cosmic Rays

The three main sources of cosmic rays are galactic, solar and terrestrial. Galactic cosmic rays are the main source of cosmic rays and provide a continuous, low flux component of the radiation environment [8]. These high energy particles are created by fusion reactions inside distant stars. Galactic cosmic rays comprise mainly of protons (85%) with small amounts of alpha particles (14%) and heavier nuclei.(1%). The energies of these particles range from almost 0 to over 10GeV.

Solar cosmic rays, which are produced by solar flairs, produce low and medium energetic solar material of low atomic weight.

Terrestrial radiation does not effect LEO satellites and is therefore not discussed.

Cosmic rays are heavily ionizing due to their charged radiation ions. An extreme case of ionization can also be observed when a high energy cosmic radiation particle strikes an atom. A cascade of particles and secondary radiation is produced which is extremely disruptive. It is very difficult to shield against cosmic rays to due to their energetic nature.

2.2.3 Solar Flares

Solar flares are caused by variations in the nuclear reactions of the chromosphere of the sun. Solar flares produce protons (>90%) as well as electrons and alpha particles [6]. Their fluxes and occurrences are hard to predict. Most of the fluence for an 11 year solar cycle is received in one or two bursts. An 11 year solar cycle can be divided up into 4 inactive years with a small number of events and seven years with a large number of events. The peak flux occurs between 2 and 24 hours after the burst and decays over a period of a couple days. These flares have been known to cause single-event upsets in electronics and damage to solar arrays.

2.2.4 Solar Wind

The solar wind is made up of low-energy protons and high-energy electrons ejected from the sun [6]. The density and velocity of the solar wind is fairly constant except, however, during a solar flare where the solar wind intensifies and can remain high for a number of weeks, see figure 2.4.

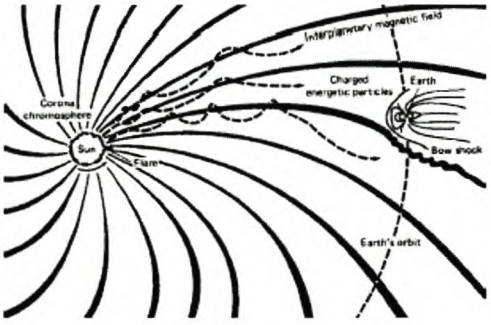


Figure 2.4: *The Earth's magnetosphere in the solar wind, showing the interplanetary magnetic field and the emanating solar wind particles ([10] Fig. 12.29A)*

2.3 Radiation Effects on Semiconductors

When energetic particles pass through matter, they lose energy through a variety of interactions and scattering mechanisms. This energy transfer from radiation to the material gives rise to the two main effects of radiation: ionization and atomic displacement. These effects cause degradation of performance in materials which might or might not be permanent.

2.3.1 Atomic Displacement

A collision between an energetic particle and silicon can result in the formation of a Frenkel defect pair, a displaced atom(Interstitial : 'I') and the vacancy left by it (Vacancy : 'V'). This is called atomic displacement. The vacancy has a tendency to recombine with impurities, while the interstitial atom, which is extremely mobile in the lattice, has a strong tendency to displace an impurity. Because the interstitial easily exchanges charge with other silicon atoms it can therefore move quickly though the lattice. A 'V' defect can also encounter another 'V' defect forming a 'divacancy' which has been known to cause electron-hole recombination which gives rise to 'leakage currents'. Neutron strikes on semiconductors have been known to produce vacancies and interstitials. A neutron strike is non-ionizing since neutrons have no charge.

2.3.2 Ionization

Energy loss from high-energy radiation results in the formation of electron-hole pairs. This a result of the valence band electrons in a solid being excited to the conduction band. If an electric field is applied then the electrons are highly mobile. Therefore any solid conducts at a higher level than is normal for that solid. The positive charged holes are also mobile, and their production and trapping cause major degradation in bipolar devices. Only small amounts of energy are required to produce an electron-hole pair (18 eV in SiO_2) [8] with no momentum transfer to atom required, therefore the energy causing the ionization is not as critically important as in atomic displacement although the number of pairs produced depends upon the particle energy. The amount of energy deposited in a material by ionization is called 'dose' and measured in 'rads'. Normally with ionization the dose-rate is specified which is the average energy absorbed per unit mass and time and termed as rads per second. Ionization is often referred to as 'total ionizing dose' (TID). Therefore TID is a cumulative long-term degradation of a device when exposed to ionizing radiation. The most serious TID effects in semiconductor devices are:

1. Temporarily lower the insulator film barrier in solid-state devices, which stops charge motion between two layers of semiconductor or conductor.

2. Freezing charge traveling across oxide into place, resulting in a semi-permanent charge sheet which effects the conductivity in layers around it.

Typical dose rates for various orbits are shown in table 2.1.

2.3.3 Transient Effects

Local ionization effects on extremely dense electronic devices can lead to a strong transient electrical response. The electron-hole pair concentration generated by ionization can

Table 2.1: *Typical Dose Rate in Various Orbits ([6] Tab.3.1)*

Orbit	Height	Inclination	Dose Rate (per year)
Low Earth	200-1000km	< 28 deg	100-1k rad(Si)
Low Earth	200-1000km	> 28 deg	1k-10k rad(Si)
Medium Earth	1000-4000km	any	100 krad(Si)
High Earth	$\approx$ 36000km	any	$>$ 10krad(Si)
Interplanetary	n/a	n/a	5k-10k rad(Si)

reach well above the doping densities of most semiconductor elements. This can cause the junctions to become 'swamped' and results in current flowing in directions which are normally blocked as well as higher voltages than were being used. These transient effects lead to a special case called 'single-event effects'.
This is a special case of ionization effect which is common in space electronics, especially in memory components. The two main classes of single-event effects are single-event upsets (soft errors) and single-event latchup (hard errors).

Single Event Upset (SEU)

A SEU, is the change of state of a bistable element caused by the impact of an energetic heavy ion or proton [8]. This change of state is not permanent or destructive and can be reset or re-written back to the state that it was prior to the upset. The upset is a result of ionization caused by a single energetic particle, or by the nuclear reaction products of an energetic proton. The ionization causes a current pulse in the p-n junction which, if larger than the required charge to change the logic state of the element (critical charge), will cause the state of the element to change. This change of state is commonly referred to as a 'bit-flip', which, in memory systems, is often observed as a logic '1' changing to a logic '0', or vice versa. Single event upsets affect both MOS and bipolar devices. The evolution in technology towards smaller device sizes leads to a decrease in the critical charge requirement which in turn increases susceptibility to SEU's. Smaller device geometries also lead to other problems such as multiple errors as a result of a single ion strike as well as rupture of oxide layers. Charge deposition is a function of a particles linear energy transfer (LET) and has the units MeV $g^{-1}cm^{-2}$. Two parameters define the susceptibility of a device to SEU:

1. Threshold LET. This is the smallest LET to produce an upset. It corresponds to a charge deposition of similar size to the critical charge amount, Q_c.

2. Saturated Cross-Section. This is when all incident ions are able to produce an upset and no upset rate increase is observed for an increase in LET. The cross-section has

the dimension cm^2 and is a function of the surface area of the sensitive node.

SEU's can also be caused by particle strikes to combinational logic. These upsets occur when a particle strikes a combinational logic node and creates a temporary voltage disturbance at that node. If the voltage disturbance propagates to a latch and occurs near the clock edge then the disturbed state may be loaded into the latch, causing the stored data to be incorrect just as if the latch itself were struck by a cosmic ray and changed state [7]. In the example shown in figure 2.5 , the correct state of the D input is a 0, but a momentary disturbance to the 1 level is latched and causes an SEU.

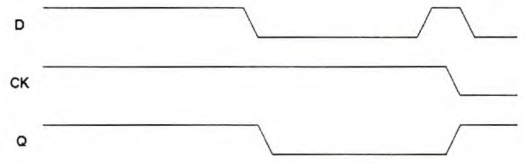


Figure 2.5: *Transient Disturbance Causing SEU in Conventional Latch ([7] Fig.1*

Single Event Latchup (SEL)

Unlike soft errors, single-event latchup is potentially destructive. In bulk CMOS devices there are parasitic vertical and horizontal n-p-n and p-n-p bipolar transistors. These parasitic transistors can form a SCR structure (silicon-controlled rectifier), which under normal conditions is 'off ' and in a high impedance state. An ion strike could turn the SCR into a low impedance 'on' state which it remains in, due to positive feedback from the p-n-p-n structure, at low voltage [8]. For latchup to occur there must be certain conditions:

1. Gain product $B_{pnp}.B_{npn}$ must be greater than 1 for positive feedback to occur.
2. Both emitter-base junctions of the transistors must be forward biased.
3. Power supplied must be able to source or sink enough current to maintain the latch.

Chapter 3

Flash Memory

3.1 Introduction

Electronic memory comes in many forms to serve many purposes. Flash memory is mainly used for fast information storage therefore being seen as more as a hard drive than as a RAM (random access memory). It is considered to be a solid-state storage device meaning that there are no moving parts, everything is electronic. Flash memory is a highly dense EEPROM (electrically erasable programmable read only memory) and is non-volatile. Non-volatile memories retain their data when power is removed where as volatile memories do not. This is attractive in space applications which have strict power budgets since power only has to be supplied to the memory during usage.

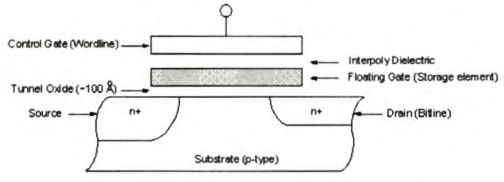


Figure 3.1: *A Typical Flash Cell ([4] Fig.33)*

3.2 How flash works

Flash memory has a grid of columns and rows with a cell that has two transistors at each intersection [19]. The two transistors are separated from each other by a thin oxide layer, about 100 Angstroms. One of the transistors is known as a floating gate, and the other one is the control gate (see figure 3.1) . The floating gate's only link to the row, or wordline, is through the control gate. As long as this link is in place, the cell has a value of

1. To change the value to a 0 requires a curious process called Fowler-Nordheim tunneling.

Tunneling is used to alter the placement of electrons in the floating gate. An electrical charge, usually 12 or 20 volts [15] (depending on the NAND structure), is applied to the floating gate (see figure 3.2). The charge comes from the column, or bit line, enters the floating gate and drains to a ground. This charge causes the floating-gate transistor to act like an electron gun. The excited electrons are pushed through and trapped on either side of the thin oxide layer, giving it a negative charge. These negatively charged electrons act as a barrier between the control gate and the floating gate. A special device called a cell sensor monitors the level of the charge passing through the floating gate. If the flow through the gate is greater than 50 percent of the charge, it has a value of 1. When the charge passing through drops below the 50-percent threshold, the value changes to 0. A blank EEPROM has all of the gates fully open, giving each cell a value of 1.

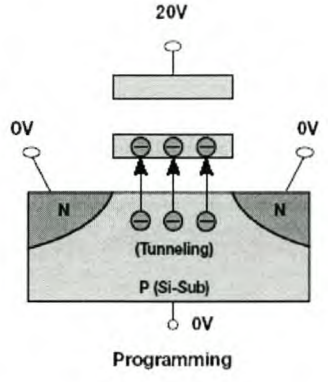


Figure 3.2: *Flash cell during programming using F-N tunneling ([18] Pg.7)*

Not all flash devices use tunneling to program. Instead Channel Hot Electron (CHE) injection is used (see figure 3.3). For this approach, the source is grounded; the control gate has the programming voltage applied to it (V_{pp}) while the drain gets approximately half the programming voltage applied to it . The voltage on the control gate is capacitively coupled to the floating gate. This turns the transistor on and allows current to flow from source to drain. Some of these electrons will have sufficient energy to pass through the oxide charging the floating gate. Electrons deposited on the floating gate charge it negatively and turns off the transistor.

Since only 3.3 or 5 volts are applied to most flash devices, the higher internal voltages required for programming and erasing are obtained using a charge pump.

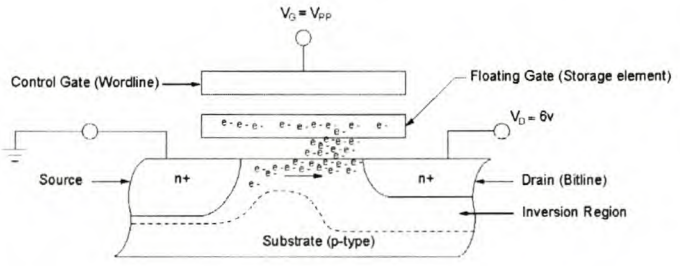


Figure 3.3: *Flash cell being programmed using Channel Hot Electron injection ([4] Fig.34)*

3.3 Nand vs Nor

There are two main flash architectures, namely NAND flash and NOR flash. The difference between the two is how the memory cells are connected (see figure 3.4). To achieve random access in flash memories, the memory cells are connected to the bit lines in parallel, as in NOR flash. The NAND flash architecture uses groups of cells (normally 16 or 32) that are stacked in series with a common bit line. Therefore NAND flash is more compact since it does not provide contacts to individual source and drain regions. However, read and write time is inherently slower because cells cannot be accessed individually; the read path goes through other cells in the stack. In order to deal with this, the device architecture divides the memory into pages. A page buffer is used to improve read time. NAND flash uses Fowler-Nordheim tunneling for both erasing and programming where as NOR flash uses CHE for programming and F-N tunneling for erasing. Because of the stacked architecture NAND flash uses a slightly higher voltage, 20V, in programming and erasing compared to NOR flash, 12V. However, the NOR memory cells wear out faster due to the CHE stress. NOR flash cells are estimated to last for 100,000 cycles where as NAND cells are estimated to last over 1,000,000 cycles.

From a practical standpoint, the main difference between the two architectures is the interface. NOR flash has a fully memory- mapped random access interface like an EPROM (erasable programmable read only memory), with dedicated address lines and data lines. On the other hand, NAND flash has no dedicated address lines. It is controlled using an indirect I/O-like interface, sending commands, addresses and data through an 8, or 16, bit bus to an internal command and address register. For example a typical read sequence consists of the following: writing to the command register the "read" command, writing to the address register 4 bytes of address, waiting for the device to put the requested data (typically 528 bytes or more) into the output register, and reading a page of data from

the data register.
The NAND flash memory space is divided as follows :

1. The smallest stored element is a byte (8 bits) or word (16 bits) depending on the bus size.
2. A row of bytes , eg 528, make up a page.
3. A group of pages form a block , eg 1 block = 32 pages. This is the smallest erasable element.
4. All the blocks together make up the entire memory area.

The two architectures allow flash cells of similar design to be used for two different purposes. The NOR architecture is suitable to be used as random access memory or bootable memory where as NAND flash is suitable to be used as mass storage memory such as a hard drive.

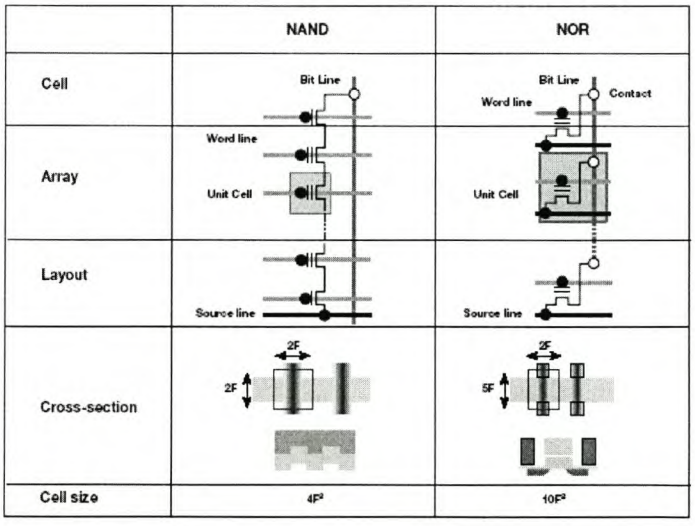


Figure 3.4: *NAND vs NOR ([18] Fig.1)*

3.4 Flash Memory Errors and their Detection

Flash memory is a highly dense and complex semiconductor device. It consists of combinational logic elements as well as memory cell elements. These combinational logic elements make up the control circuitry internal to the flash chip. This circuitry is responsible for the complex control and timing requirements needed for writing, erasing and reading from the memory cell elements of the flash chip. Therefore the high internal complexity and density make flash chips vulnerable to different types of errors, not only

from induced radiation but also from internal degradation such as wearing out of flash cells.

The charge pump has been linked to causing most of the failures associated with radiation dosage according to [11]. It is responsible for supplying the higher voltages required for erasing, writing or reading, and achieves this by utilizing internal floating-well elements. The generated output voltage is proportional to the number of charge pump stages and so when the ionizing dose of radiation shifts the threshold voltage, V_{te}, of each pass transistor, the result is a reduced V_{out}. Any stage-to-stage increases in leakage would also reduce the charge pump output. Using an external charge pump would solve this problem but NAND flash chips do not allow this option and therefore cannot be considered further as a mitigation solution.

All internal elements of a flash device are exposed to radiation, and can cause different types of upsets or errors. The main errors that can occur in flash memory devices are single event upsets, single event functional interrupts, single event latchup and bad block development. The resulting effects of these errors and possible mitigation solutions are investigated below.

3.4.1 Single Event Upsets (SEU)

Due to the transient effects of radiation in semiconductor devices (see 2.3.3), SEU's can occur in flash memory devices. If left unprotected, these soft errors can corrupt the data in such magnitude that the data becomes unusable. Therefore measures must be taken to protect the data stored in the flash memory system.

Since SEU's are soft errors, no permanent damage is done to the physical memory, the main concern is corrupted data. Data can basically be protected from corruption either by encoding the data or by storing multiple copies of it. The latter option is unfeasible for a satellite memory system due to the size and power restrictions that are common in satellite specifications. Therefore encoding the data is the obvious choice. This entails taking the received data and encoding it in a way that monitors the status of the data and if an error occurs the code will detect the error and correct it if possible. The various schemes and implementations are investigated in section 4.

3.4.2 Single Event Functional Interrupt (SEFI)

Single event functional interrupts are more complex than simple SEU's. SEFI's occur as a result of transient errors occurring in the control circuitry in the flash chip. This

causes the flash chip to malfunction and become unpredictable. In SEU's the error can be identified and corrected and in many cases the upset cross-sections are representative of the geometrical areas of the sensitive regions. However, in SEFI's, the event is caused by a SEU at a sensitive section of a microcircuit device to which there is no direct access. Since the exact location of the area cannot be identified, we can only observe the failure of the device function.
There are two main types of SEFI's that occur: regular SEFI's and irregular SEFI's. Regular SEFI's are easier to detect and have some of the following characteristics :

1. Operation of the flash chip suspends.
2. The operating current becomes slightly higher, not as high or as dangerous as latchup.
3. Can occur with a low linear energy transfer (LET).

With SEFI's the control circuitry on the chip is effected and can cause the memory chip to malfunction or suspend. These error states can occur during read, write or erase procedures. The different types of SEFI's and their respective possible mitigation solutions are as follows [12]:

1. Block erase SEFI : While erasing the flash device's ready signal never exits the busy state and therefore will not complete the erase which suspends operation. A suggested mitigation procedure is to reset the power to the flash chip and repeat the erase.
2. Partial erase SEFI : The block erase procedure suspends at the first address but few blocks are erased. To mitigate just repeat the intended erase process.
3. Write SEFI : The write operation's complete status suspends, therefore causing the controlling program, waiting for the complete status, to also suspend. Resetting power to the flash chip should mitigate the error.
4. Read SEFI : During a read process the sensing circuitry suspends due to a charge buildup. The next read operation does not clear the errors and will therefore return corrupt data. By repeating the intended read process or by cycling the power to the flash chip, the error can be mitigated.
5. Irregular SEFI's : There are two main irregular SEFI's that occur and are:
   (a) Irregular Read SEFI : The read operation locks in and endless loop, reading out the same data over and over which corrupts the intended output. This can be reset by cycling the power to the flash chip.

(b) Irregular Write SEFI : The write operation stops and suspends. Resetting the power alone wont initialize the device to a pre-SEFI state until the previous write operation is restarted.

As mentioned earlier, the current drawn by a flash chip increases when the chip is irradiated. The current variations observed for different flash chips [11] are shown below in table 3.1.

Table 3.1: *Flash Chip standby currents after being irradiated*

Flash Chip	**0 Krad**	**12 Krad**	**16 Krad**
Intel 28F320 (biased*)	0.275mA	0.4mA (non-func)	
Intel 28F320 (unbiased**)	0.275mA	0.3mA	0.35mA (non-func)
	0 Krad	**2 Krad**	**15 Krad**
Intel 28F640 (biased)	0.25mA	4mA	
Intel 28F640 (unbiased)	0.25mA		1mA (spec-limit=0.9mA)
	0 Krad	**20 Krad**	**120 Krad**
Samsung km290128 (biased)	0mA	0.7mA	
Samsung km290128 (unbiased)	0mA	0mA	0.55mA (spec-limit=0.05mA)

*Biased meaning static biased, where power is supplied to the chip during irradiation but no address cycling or data access operations are performed.**Unbiased meaning that all pins are grounded during testing.

There are obviously two current values to monitor, the operating current and the standby current. The memory system is only operational while storing images or reading out the stored images, therefore spending most of the time switched off or in standby mode. The standby current is therefore a good indication of the total absorbed dose status of the flash chip. As seen in table 3.1, the standby current increases as the total dose absorbed by the flash increases. Monitoring this current can help predict SEFI's and other radiation effects.

3.4.3 Single Event Latchup (SEL)

Single event latchup is the most destructive of flash device errors, see 2.3.3. It is a destructive condition that can destroy the device if current is not limited or removed within allowable time. The operating flash current has been observed to rise from the expected value , 20mA, to 430mA. The way to detect latchup is to monitor the current into the flash. The current levels must be distinguished between SEFI's and latchup, however both errors are treated similarly.

When latchup is detected the power to that chip must be cycled. When reset, the power levels should return to normal. If the power levels continue to exceed acceptable levels then power should be permanently removed from the device and its failure reported.

3.4.4 Bad Blocks

Invalid blocks are defined as blocks that contain one or more invalid bits whose reliability cannot be guaranteed. Invalid blocks have the same AC, DC characteristics as valid blocks and do not effect the performance of valid blocks. This is because they are separated from the bitline and common-source line by a select transistor.

Nand-flash devices off-the-shelf can have a certain number of bad blocks. Bad blocks (invalid blocks) can also develop over time due to radiation or overuse. Using a bad block could result in invalid information being stored or read. Therefore hardware measures should be taken into account to mitigate the effects of bad blocks. If a block has become unreliable and marked as a bad block then it cannot be recovered or used reliably again. Due to the finite number of cycles which a block can be read/programmed/erased, after extensive use more and more blocks will become invalid.

Prior to shipping manufacturers usually mark which blocks are invalid on the chip. For instance, Samsung marks the 6th byte in the spare area of the first two pages of each block with *FF*H if valid. If the value of this byte is not *FF*H then the block is invalid. Care must be taken on first use because the bad blocks are still erasable and the initial bad block data could be lost. The device should therefore be checked upon initial use as shown in figure 3.5 and the data recorded.
Since bad blocks develop during use, they must also be detected. This can be achieved by reading the status register following an erase or programming operation. Flash devices provide a status byte of information which is updated after every operation and can be read from the status register in order to check whether the operation performed was successful or not. Each bit in the byte represents a certain status check and is flagged accordingly. Not all flash devices are exactly the same in this regard but are all fairly similar. If an erasure or program is unsuccessful then the block is likely to be invalid and should be replaced accordingly, although a verify-after-program status error might only need ECC correction due to the read error and not the actual program error, see table 3.2. An error following a read operation does not require block replacement and should be corrected by the implemented error correction code. As mentioned previously, when a new block is detected as being bad it should immediately be replaced and then prevented from further use. Block replacement is done by reprogramming the data, initially programmed

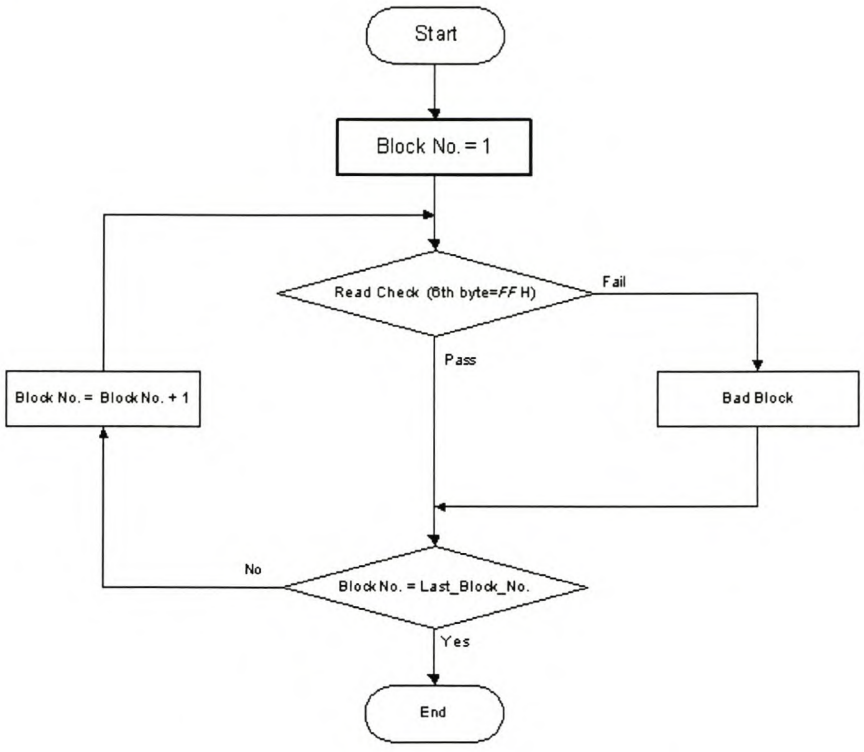


Figure 3.5: *Bad Block Test Flow Diagram*

Table 3.2: *Mitigation solutions for operation failures*

Erase	Erase failure	block replacement
Program	Program failure	block replacement(ECC is a verify error)
Read	Bit failure	ECC correction

into the bad block, into a new block and then continuing the programming process. The system should then be prevented from again using that bad block. This can be achieved by creating a bad block table or by using another appropriate scheme.

3.5 Radiation Test

To investigate the effects of radiation on a flash memory device it was decided to design a test board which would later be radiated. The results of the test would then give insight on the various requirements of this study.

The test was run over a period of three days, radiating the board with **Co-60** ions. However, the experiment was only run till 5pm each day and then suspended overnight.

This gave an insight as to how the flash devices would react if given time to recover from the radiation.
Table 2.1 shows that a satellite in LEO can experience a total dose rate of up to 10kRad per year, therefore a satellite with a lifespan of 3 years can obtain a TID of 30kRad. It was therefore chosen to radiate the flash devices at a tempo of 2,5 kRad per hour until a TID of 35kRad was reached. At this value significant results were obtained.

Firstly, in this section, the board design will be discussed, followed by the radiation test program design and then finally the results obtained by the test.

3.5.1 Radiation Board Design

Since the board was to be radiated and therefore risking permanent damage, it was decided to design a board which was simple enough to perform the required tasks without being too expensive. The basic needs of the radiation board was :

- One or more flash devices with which to run the tests on. The actual flash device chosen was not critical since all flash devices have the same package and pinout, therefore can be chosen at a later stage.
- An intelligent controller with which to perform the radiation test program and interface the flash devices. The controller would also have to be able to interface to a computer to report the results of the test.
- In case of a power cut to the board being radiated, the controller would also have to be independently reprogrammed with the radiation test program.

From these requirements the following choices were made regarding the radiation test board:

It was decided to implement two flash devices on separate busses. This enabled both flash devices to be operated in the radiation test which gave an interesting view on how exact same model flash devices are effected differently by similar radiation. It was chosen to use **SAMSUNG K9K2G08U0M** flash devices since they were the largest flash devices available at the time of testing. These are 256 Megabyte flash devices which actually consist of two 128 Megabyte flash chips inside the device.
The intelligent controller was realised using an **ALTERA CYCLONE EPIC3T144C7** FPGA. This was a suitable decision since this FPGA meets all the requirements: it was within the price budget, has enough pinouts to interface both flash devices, is large enough within which to implement the radiation test program and also fast enough to meet any timing requirements.

An **ALTERA EPC2LC20** programming device was implemented to reprogramme the FPGA immediately as power is applied. This was in case any of the devices stopped responding and required the power to be removed from the entire board.
A simple **serial interface** was implemented as external hardware to the board as it was unsure at the time how communication was going to be implemented. The serial device was connected to the FPGA via spare input/output pins. The serial interface was mainly used as a data dump which was logged by the computer for assessment after the entire test was completed.

Figure 3.6 shows a picture of the board. The schematic of the test board is shown in appendix A.1 and the layout of the printed circuit board is shown in appendix A.2.

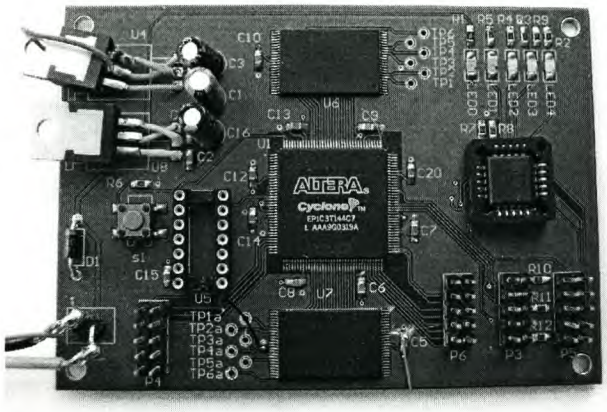


Figure 3.6: *Radiation Test Board*

3.5.2 Radiation Test Program

The main goal of the test program was to test the flash device's operation under a heavy radiation stress environment. To test a flash device, it must be written to then the results verified. But in order to write to flash memory, it must first be erased. Another goal was to monitor when blocks become invalid and to log their position in the flash memory. This was achieved by implementing a temporary bad block table.

The flow diagram for the test program is shown in figure 3.7. Testing began by first initializing the temporary bad block table which clears all information from the previous test cycle. Next the entire chip was erased. While in this state, each single erase process was timed and the longest erase per cycle recorded. Also while erasing, the status register of the flash device was constantly checked to verify the operation and detect blocks going

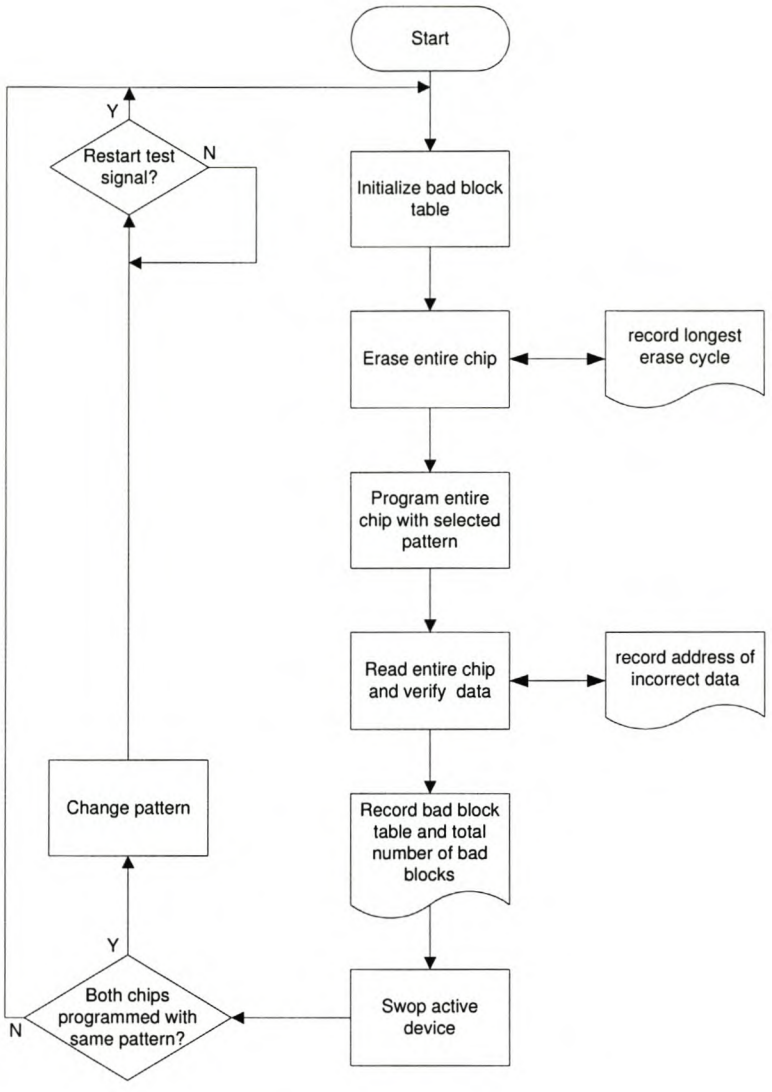


Figure 3.7: *Radiation Test Flow Diagram*

bad. If an operation was unsuccessful then the bad block table was updated. The next step was to program the entire device with a certain pattern. It was chosen to program a checkerboard pattern, alternating 1's and 0's, to test the flash device. This was suitable because it checked every single bit during a test. Both chips were programmed with the pattern then it was changed so that all the previous bits that were programmed to 1's were then set as 0's and vice versa. As with erasing, after each program operation the status byte was checked to verify the operation. After programming the entire contents of the flash was then read and verified. The address of any corrupted data was recorded by the PC. Finally, the contents of the bad block table was outputted and recorded as well as the total number of bad blocks. Once a test cycle was completed, the active flash

device was swopped to the other device and the test rerun. If both flash chips had been run with the same program pattern then the pattern was changed and the test suspended until signalled to run again. Therefore the test cycle was run twice, one for each chip, then suspended.

3.5.3 Results

The results for the maximum erase time per cycle versus total ionizing dosage, explained in 2.3.2, was plotted and shown in figure 3.8. One can see a definite difference in erase time between the two identical flash chips. As expected, the erase times increase with an increase in TID. The two vertical blue lines indicate when the experiment was suspended overnight. A definite decrease in erase time can be seen at the first interval while no definite effect can be seen at the second. It is apparent that the flash device 'recovers' a degree from the first round of testing but does not after the second round.
The resulting timing values are shown in table 3.3. Both erase times increased by a small amount, tens of microseconds. This is not a major increase and therefore shouldn't play a major role in monitoring the flash device's behavior other than influencing a watchdog timer value that monitors operation completion.

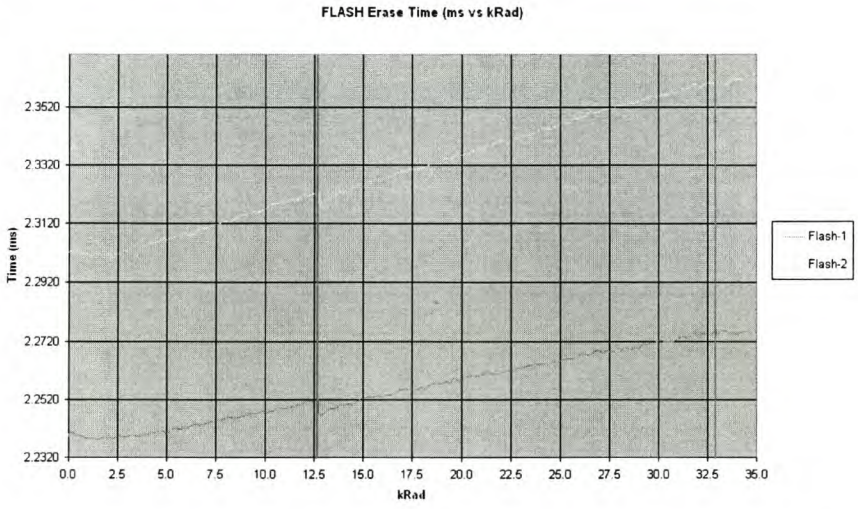


Figure 3.8: *Maximum single erase operation duration per cycle vs TID*

Table 3.3: *Erase time results*

	Begin (ms)	End (ms)	Increase (us)
chip 1	2.2393	2.2769	36.7
chip 2	2.3023	2.3601	57.8

The results for the bad block occurrence are shown in figure 3.9. The figure shows four graphs, this is due to the fact that each of the two 256 M-Byte devices have two 128 M-Byte chips inside it, as explain in 3.5.1. *Si-1* and *Si-2* represent the two chips in flash device 1, similarly, *Si-3* and*Si-4* the two inside device 2. Each 128M-Byte chip has 1024 blocks, non of which were invalid before the test began.
The first bad blocks started occurring at a TID of 16.918 kRad, after which they began to increase linearly with increased dosage with a maximum tempo of roughly 150 bad blocks per kRad. As explained in the erase-duration test, the two vertical blue lines indicate when the experiment was suspended and left over night. The second interval shows a distinct decrease in the number of bad blocks for 3 of the 4 chips. *Si-4*, which had already almost reached total failure at that stage, showed no improvement however the other three each recovered over 239 blocks that were previously invalid. With resumed dosage, bad blocks occurred as they had before, at similar rates.

These results confirm the need for a bad block managing system to prevent data loss which would occur if unprotected. The fact that invalid blocks can become valid again shows potential for a bad block recovery process. These findings will greatly influence the concept designs at a later stage.

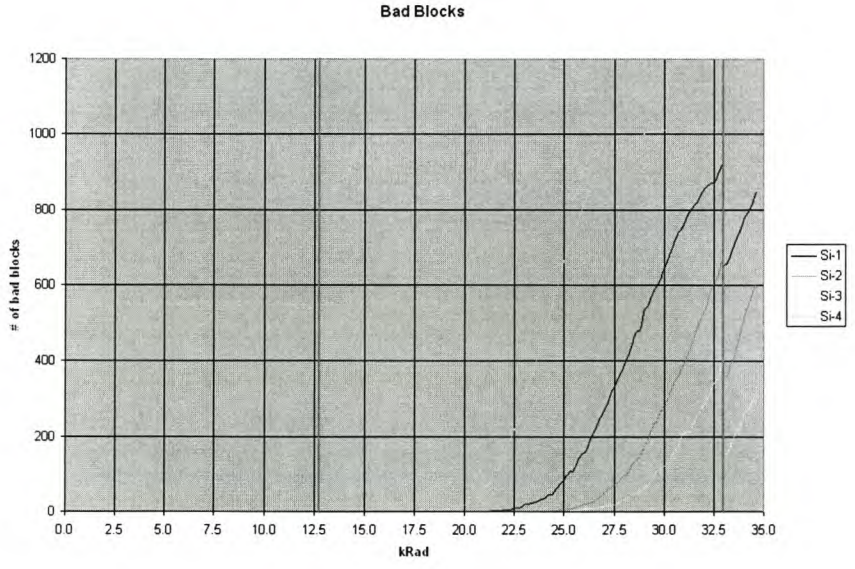


Figure 3.9: *Total bad blocks vs TID*

Chapter 4

Error Detection and Correction

Due to the high possibility of soft-errors occurring in the flash memory system, caused by radiation, it is necessary to implement an error detection and correction (EDAC) system to decrease or eliminate these errors. The number and position of soft errors occurring in the flash memory is hard to predict and therefore complicates the choice of EDAC system and how to implement it.

Most EDAC systems add extra information (redundant bits or bytes), someway, to the data that is being stored to be able to detect errors if they occur. A greater level of protection leads to increased amounts of redundant information being created. This redundant data also needs to be stored and protected from soft-errors therefore decreases the available memory space for actual data. A tradeoff analysis must be done to determine the minimum level of protection required (eg: 1 bit correctable per byte or 2 bit correctable per byte, etc) versus the maximum amount of redundancy that is acceptable (eg: 20% or 33% redundancy).

Different EDAC systems provide different types of protection, such as protecting bits in a byte or bytes in a word. There are also different ways of implementing each EDAC system, either as software or as hardware. The main factors that influence the choice of EDAC implementation are:

1. Required level of protection and redundancy.
2. Speed of encoding or decoding
3. How much data to encode or decode
4. How data is stored and in what medium
5. Cost of EDAC.

Firstly software and hardware EDAC systems will be investigated and then a suitable EDAC system will be chosen accordingly. Since SUNSAT will be storing raw image data and is in a low earth orbit, the level of protection required will be assumed to be 1 bit error per byte correctable and 2 bit errors per byte detectable.

4.1 Hardware vs Software EDAC

The chosen error correction code (ECC) can either be implemented using hardware, software or a combination of the two. Hardware being a dedicated hardware device, such as a field programmable gate array (FPGA) programmed to implement the chosen ECC, and software being a subroutine run by the on-board computer (OBC). The advantages and disadvantages of using the above implementations are explored in this subsection and a suitable implementation is chosen.

4.1.1 Software EDAC

Software implemented EDAC systems can be implemented on pre-designed systems that do not have hardware EDAC. With a software EDAC, the data is stored unencoded in memory. The EDAC software routine then accesses the data and creates the corresponding ECC and stores it in alternate memory or in alternate parts of the same memory. 'Scrubbs' are then performed periodically and data updates are performed, if required. A 'scrubb' being a check on random data to ensure that the data is correct [16].
The advantages of using software EDAC are:

1. ECC generation can be done after the streaming data is stored therefore does not have to be generated on-the-fly as the data is captured. This relieves the strict timing constraints that comes with on-the-fly encoding.

2. No extra hardware is required in implementing the EDAC system therefore software EDAC is cheaper than hardware EDAC.

The disadvantages of software EDAC are:

1. The requirement of a microprocessor to run the ECC sub-routine means that the OBC will have to be used. This results in loading OBC which could be needed by other, more important, processes.

2. The EDAC will not be as reliable as in hardware implementation since only periodic 'scrubbing' is performed.

3. The redundant ECC information has to be stored after the intended data has already been stored. This could prove difficult to do since flash memory has to be erased

before writing to it and erasure can only be performed in blocks. Therefore to update data in the flash memory the entire block has to be temporally stored elsewhere, updated and then written back to the flash memory.

4.1.2 Hardware EDAC

Hardware EDAC is performed by a dedicated hardware device implemented especially for that purpose. An EDAC system can easily be implemented using a FPGA or ASIC (application specific integrated circuit). As the intended data arrives it is passed to the ECC encoder which performs the EDAC. The encoded data can then be stored in the memory system as required. On retrieval the encoded data is passed to the decoder which then outputs the decoded data to the required recipient.
The advantages of hardware EDAC are:

1. No extra processing is required by the OBC therefore not a burden on the rest of the system. The EDAC system can be seen as being transparent to external processes.
2. Hardware EDAC can cope with the strict speed constraints and therefore on-the-fly encoding and decoding can be performed.

The disadvantage of hardware EDAC is:

The extra cost, weight, power and design of implementing additional hardware. More hardware also increases the possibility of radiation errors since there are more susceptible components to radiation.

4.1.3 Conclusion

Nand-flash memory is the chosen storage medium therefore random access to an individual memory location is time consuming and tedious. This fact makes software coding unattractive due to its need to access random bytes. Loading the OBC is also a downside of using software EDAC since it is desirable to make the EDAC processes transparent to the rest of the system. These problems can be overcome by using hardware EDAC. The extra cost and power of implementing the hardware is minimal compared to the benefits obtained. **Hardware EDAC** is therefore the obvious choice.

4.2 Error Correction Codes (ECC)

4.2.1 Hamming Codes

Hamming code is a simple parity check code, the hamming rule is $d + p + 1 \leq 2^p$ [20], where d is the number of data bits and p is the number of parity bits. Therefore to

implement a 1 bit-per-byte correctable hamming code, 4 redundant bits have to be added per byte. This results in a (12,8) hamming code. The first number, 12, being the total number of bits in a codeword and the latter, 8, being the number of actual data bits. The redundancy efficiency of this code is $8/12 \times 100/1 = 66\%$. Therefore 33% of the memory will be used as hamming check bits which is quite high. The capability of the hamming code will be 1 bit correctable and 2 bit detectable. This is sufficient to meet the EDAC requirements. The correctable rate is therefore 1 bit per byte giving 12.5%.

The speed of the incoming data can reach 12 Mbytes per second. Data can be written to flash at roughly a maximum speed of 20 Mbytes per second. Therefore since flash stores information in the form of bytes it would be efficient to combine 2 data bytes' parity bits (4 bits each) to form a byte and then store that byte after the two data bytes into the memory. So for every 2 data bytes received, 3 bytes are written to memory.
If the maximum arrival speed is considered, 12 Mbytes per second, a new byte arrives every 83ns. Therefore the maximum time allowed to write 3 bytes to memory is 166ns, or 55ns per byte. Since the flash can be written to every 50ns and that the hamming check bits are generated instantly, this hamming technique will be sufficient to meet the timing requirements. The timing constraints can also be relaxed somewhat by using buffering.

The decoding process will be similar to the encoding. Three bytes will be passed to the decoder which will output the two corrected data bytes.

The creation of the check bits, or redundant bits, requires parity checking algorithms. These algorithms are simple to install and can easily be implemented using any standard FPGA.

4.2.2 Rectangular Parity Codes

This error code creates vertical and horizontal parity bits which protect a code block of several bits (see figure 4.1). If a bit flip occurs, such as data bit D_5, then two parity bits will be in error, P_5 and P_2, showing the location of the error bit. This method would not be practical and wont be pursued further for the following reasons :

1. For the level of protection required, one bit per byte, the redundancy would be too high($> 100\%$).

2. The process in which the parity bits are calculated and stored would require random access to the nand-flash memory. This process would be extremely slow and the overhead would be too high to make it feasible.

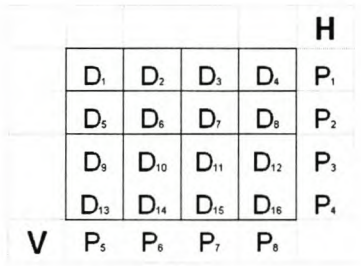


Figure 4.1: *Rectangular Parity code*

4.2.3 Reed-Solomon Codes

This is a cyclic code which creates blocks of symbols called *codewords* (see figure 4.2). The symbols can be any number of bits in length (M), for this project the symbols width will be chosen to be eight bits wide to correspond with the flash bus width. Reed-Solomon codewords are specified by the total length of the codeword, i.e. the number of symbols (N), and the number data symbols (K). The number of redundant check symbols (R) is therefore equal to $N - K$. The maximum size of the codeword block is dependent on the symbol width as follows: $N_{max} = s^m - 1$ [5].

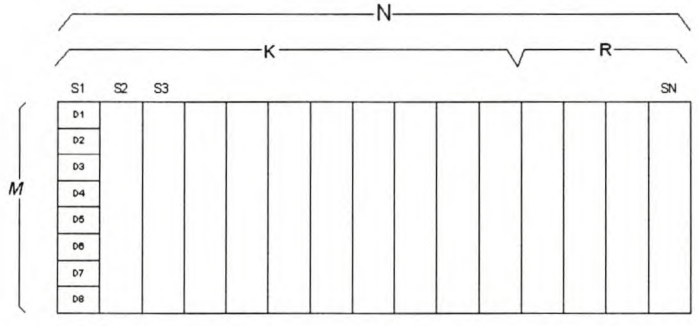


Figure 4.2: *Structure of a Reed-Solomon codeword*

Reed Solomon encoding is done on-the-fly, meaning that the encoding process occurs as data arrives at the encoder and is passed through. As with other code forms, reed-solomon codes can only correct up to a certain amount of errors per codeword, but in this case errors are not bit errors but symbol errors. This means that reed-solomon codes can correct up to a certain amount of symbols per codeword independent on the number of errors inside the symbol. So, in this chosen case, whether one bit is in error or all eight bits are, the entire symbol will be corrected and seen as one corrected symbol. It is easy to see the benefits of using reed-solomon coding, in that single bit upsets or multiple bit

upsets can be corrected. The maximum amount of correctable symbols per codeword, (t), is given by the formula $t = (N - K)/2$.

As with a hamming code, a reed-solomon encoder-decoder can also be easily implemented in an FPGA. The reed-solomon encoder-decoder is however more complex and therefore requires a larger FPGA, depending on the specifications of the chosen code. Creating and programming a reed-solomon encoder-decoder from scratch in VHDL is outside the scope of this thesis but can be easily programmed into an FPGA using a *megafunction*, explained in 5.2.5, which is available to certain FPGA's such as the *Altera Cyclone* FPGA.

To compare the implementation of a proposed reed-solomon code to a hamming code a sample calculation was done as follows:

1. Since the chosen symbol width, M, is chosen to be eight bits wide, the maximum block size is therefore: $N_{max} = 2^8 - 1 = 255$ symbols.

2. To compare to the previous hamming code assume a redundancy of 33%. Therefore $R = 33/100 \times 255 = 84.15 = 84$ symbols. The number of data symbols is therefore 171.

3. The number of correctable symbols is as follows: $t = (255 - 171)/2 = 42$ symbols. Therefore 42 of the 171 symbols can be corrected, this gives a correctable rate of $42/171 \times 100/1 = 24.56\%$, which is better than the hamming correctable rate of 12.5%. However, that is a best-case scenario, if single bit errors occur all in separate bytes the correctable error rate can drop from 24.56% to as low as 3%.

Comments: Although the correctable rate of this reed-solomon example is higher than the hamming code example, the hamming code can correct 1 error in every byte where the reed-solomon code can only correct 1 error in 42 of the 171 code bytes or symbols if they occur in that manner. So there is a tradeoff between a code that can correct multiple bit errors or single bit errors.

4.2.4 Convolutional Coding

Convolutional coding creates symbols using shift registers and modulo2 arithmetic. Basically the encoding process is as follows: for each incoming bit a symbol is created which is then transmitted and stored. To decode, the symbol stream is retransmitted to the decoder where they are temporarily stored and decoded using trellis diagrams and mapping.

The redundancy for convolutional coding is high (50%) because for every bit received a codeword is created, usually 2 bits. The decoding as also slow and tedious due to the

trellis diagram generation and mapping. Convolution coding is therefore not suitable for a high-speed, low redundancy storage medium.

4.2.5 Polynomial Codes

Polynomial codes treat strings of binary information as coefficients of polynomials x^{k-1} *to* x^0 of degree 'k-1'. Eg: the binary string "101" is represented as $1x^2 + 0x^1 + 1x^0 = x^2 + 1$. A generator polynomial, $G(x)$, must also be chosen which has both highest and lowest order bits as '1s'. The process for choosing a suitable generator is complicated and not discussed further.

The encoding process is fairly simple, the information bits to be encoded are placed as coefficients in a polynomial string, $M(x)$. r zero bits are then appended on the end of $M(x)$ forming a new polynomial, $Q(x)$ which is then divided by the generator polynomial, $G(x)$. The remainder is then subtracted from $Q(x)$ to form the codeword $C(x)$ which is then storable. Therefore all stored codewords are divisible by the generator polynomial. The degree of protection is controlled by the complexity of the generator polynomial.

Decoding is similar to encoding, the codeword is divided by the generator polynomial and if no remainder occurs then the data is assumed to be correct. If a remainder occurs then the data is assumed to be in error and corrected by analyzing the remainder. To compare the polynomial code to the previous codes an adaptation is done as follows:

1. The data packet size is chosen as 8 bits therefore k must be 8.

2. The position of a possible error in the codeword must be located, knowing that the possible position of the error is greater than k, due to the redundant bits been added on. Therefore, setting r as 3 gives only a possible 8 positions, which is not enough, therefore r must be chosen as 4, giving 16 possible positions.

3. The codeword is now 8 bits + 4 syndrome bits equalling 12 bits in total. This gives the same redundancy as the hamming code counterpart.

The polynomial code is equal to the hamming code in redundancy and correctable rate. The problem with this code is in the implementation process. Encoding and decoding requires polynomial division and to implement it in hardware requires modulo2 arithmetic of a k stage shift register which is too time consuming for the bandwidth required so therefore not a viable solution. So unless modulo2 arithmetic could be done in parallel it is too slow for the required purpose but even so, hamming code is easier to implement and gives the same results and therefore would be better choice.

4.3 Conclusion

The above investigation singles out two error correction codes that meet the requirements of high-speed, low redundancy storage medium, namely the Hamming Code and the Reed-Solomon code. Both are easily implementable using minimal extra hardware such as an FPGA. The adapted hamming code discussed in 4.2.1 gives a correctable error rate of 12.5% with a redundancy of 33.3%. The adapted Reed-Solomon code of equal redundancy discussed in 4.2.3 gives a correctable error rate of 24.56% which is almost double that of the hamming code. But the Reed-Solomon correctable error rate is dependant on where the errors occur and can drop as low as 3% as a worst case scenario. Hamming code can only correct single bit errors in a byte and detect double bit errors where Reed-Solomon can correct multiple adjacent bit errors.

Therefore the main factors determining which ECC to implement is what type of bit flips are expected to occur and if burst errors will be experienced during reading or programming with the flash memory.

Hamming error correction code is fairly common among high speed storage medium and does not present much of a challenge whereas Reed-Solomon coding is far more complex and has a wider protection possibility and will therefore be chosen as the error correction code.

A **Reed-Solomon** coding scheme will therefore be designed and, according to 4.1.3, implemented in **hardware**.

Chapter 5

Designing the Memory Protection Unit (MPU)

The main errors that can occur in the flash memory were discussed at the end of chapter 3. The detection, mitigation and reporting of these errors is the main focus of this thesis and forms the basis from which the memory protection unit will be designed. Throughout the rest of this thesis the overall system that monitors, detects, mitigates and reports errors in the flash memory will be referred to as the memory protection unit, or MPU. This design is focused on monitoring one flash device, although it is known that most memory systems consists of arrays of devices. Therefore, throughout the study, a modular approach will be followed so that parts of the overall design can be duplicated to monitor more than one device.

This chapter begins by discussing the basic requirements of the MPU. Using these requirements, decisions are then made concerning the various subsystems and their respective implementation strategies. Concept designs are then developed for each subsystem which is later refined, detailed and finally added together to form an overall concept design for the MPU system.

5.1 Specifications Required by Project

In order to reach a final goal, certain rough estimates concerning the specifications must be made. The random nature of radiation induced errors in a memory system makes deciding upon exact specification difficult but imperative to build a framework in which to design.
The main requirements for the MPU are:

1. The data stored in flash memory must be protected to a minimal level of one bit correctable per byte stored.

2. The flash memory device must be protected from single event latchup and any other destructively high current events.

3. The MPU must be versatile to handle any flash device of any size.

4. The CCD camera would provide synchronous 8-bit data at a maximum frequency of 10 Mbytes per second to be stored in the flash memory. The data would also be accessed from the memory at up to that same speed. The MPU must therefore be able to function at that speed.

5. The flash memory device must also be protected from functional errors.

6. The power consumption of the MPU must not exceed 1Watt.

7. Faults and errors must be reported to the OBC even if the flash memory unit is offline or damaged.

8. The MPU must be modular and independent from the actual mass memory unit controller and driver.

5.2 Developing a Concept Design

From the requirement analysis, a conceptual design must be created in order to propose a way to meet and satisfy the needs set out. Evaluating the requirements provides insight into the challenges ahead and gives ideas on how to solve these issues. Firstly the basic subsystems that make up the overall system must be chosen, giving structure to the broad outline. The engineer must then contribute innovations to come up with a concept on how the system must be built. This modular process is then repeated each time, in more detail and on a lower level, until the final design is implemented.

In the following subsections, the concept design of each subsystem is investigated and discussed using the insight acquired from the requirement analysis and from the constraints set by the peripheral systems that interact with the mass memory unit.

5.2.1 Scope and initial problems of design

It is important to define the scope of the thesis in order to focus the study on what is relevant to fulfil the requirements and not on areas that fall outside of the scope. This study involves the radiation errors associated with flash memory, not the design of a mass memory flash system itself. Therefore certain aspects of the system must be addressed such as:

1. The design of the actual mass memory unit, or RAMDISK, is not part of this study and therefore not undertaken. This includes aspects such as:

   (a) Interfacing with the peripheral subsystems such as the on-board computer, imager and telecommand systems.

   (b) Architecture of the RAMDISK. This includes how the RAMDISK is physically structured and how the flash memory system is interfaced.

   (c) How power is supplied to the RAMDISK.

2. The peripheral subsystems that interact with the mass memory module, such as the imager, telecommand and telemetry system, and power regulation units were assumed as given entities which comply to their specifications.

3. The actual type flash chips used in the mass memory module will be assumed to be NAND flash. This is a logical decision since NAND flash is perfect for the requirements of a low power high density mass storage medium and NOR flash is more suitable for random access memory, as discussed in section 3.3.

To be able to begin designing the various subsystems certain assumptions and choices must be made in order to narrow possibilities and focus on an implementation scheme. These assumptions and choices are:

1. Choosing and implementing the EDAC code, decided upon from the tradeoff analysis performed in chapter 4, largely depends on the actual flash ram chip for which it is being implemented. The operation of different flash chips is fairly similar and therefore not of great concern. It is the internal architecture of the memory area in different flash chips that varies considerably. This includes different page sizes and block sizes which influences how data is stored and how areas are addressed. These changes greatly influence the codeword size of EDAC codes and are therefore of great importance.

   The K9F1208X0A, made by Samsung, was selected to be the memory chip in the design. It was chosen due to its high density, low power and sufficient availability. Although the design will be made specifically for this model, it should be easily updateable to accommodate a different size of NAND flash chip. This was necessary to ensure the design does not quickly become redundant due to the constant improvement of flash chips. The K9F1208X0A is discussed closer in subsection 5.2.2.

2. The imager is modeled as a 8-bit data pump which outputs the data bytes at a maximum speed of ten megabytes per second. The speed of the bytes are however allowed to fluctuate to any slower speed. The outputted data from the imager is

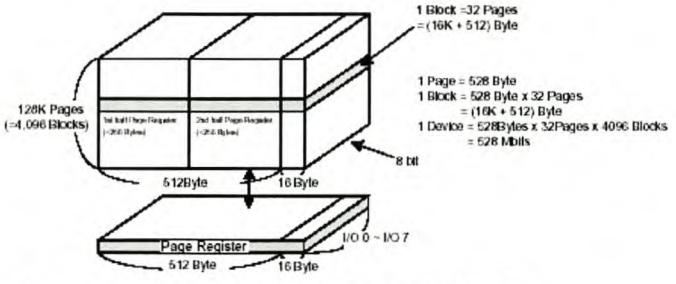
accompanied by a strobe line which informs the memory unit of the arrival of a new byte. No indication of the size of incoming data is given only that it will output a certain amount and then stop. The imager is assumed to be controlled by the OBC and that the memory module does not interact with it, merely collects the outputted data.

3. A regulated 3.3 Volt power supply is assumed to be supplied to the mass memory module and therefore available to the MPU. A later development required a 1.8 Volt supply and therefore also assumed available.

5.2.2 The K9F1208X0A Flash Memory Module

The entire thesis is based around flash memory devices. Therefore, in order to understand the device drivers better, it is imperative to have a closer look at the chosen memory device. This helps make critical decisions when designing subsystems that have to communicate with the memory devices in some form.

According to the datasheet [14] the K9F1208X0A is a 528M-bit (64M-byte) non-volatile flash memory component. The memory is organized as 131,073 rows (pages) by 528 columns, see figure 5.1, with a bus width of 8 bits. The memory array consists of 4,096 separately erasable 16K-byte blocks. Addressing the entire memory space requires a 26 bit address, therefore 4 cycles are required for byte-level addressing.



	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	Column Address
2nd Cycle	A8	A9	A10	A11	A12	A13	A14	A15	Row Address (Page Address)
3rd Cycle	A16	A17	A18	A19	A20	A21	A22	A23	
4th Cycle	A24	"L"	"L"	"L"	"L"	"L"	"L"	"L"	

Figure 5.1: *The K9F1208XOA Memory Array Configuration ([14] Fig.2-1)*

Electrically the maximum operating current is 20mA and typical supply voltage is 3.3V. This gives a low power consumption figure which is favorable. The minimum number of valid blocks is 4,026 of the 4,096, therefore a maximum of 70 bad blocks, to begin with, is assured. The first block however, placed at 00h block address, is fully guaranteed to be a valid block therefore would be a suitable place to store valuable data.

From these specifications it was noted that:

1. The 8-bit data width of the flash memory coincides with the width of the data stream outputted from the imaging subsystem which should prove useful in meeting the timing requirements caused by streaming data.

2. The page size is 528 bytes which will have a large influence in choosing a codeword size since it would be favourable for each codeword to fit exactly into a page without wasted space or overflow.

3. Having to send a 4 cycle address each time a specific page is addressed will influence the addressing scheme due to the extra time required.

4. The non-volatile nature of flash memory allows all power to be removed from it without losing any of its data. This will prove useful since flash memory is less susceptible to radiation damage when not powered.

5.2.3 System Level Layout

To begin the design process, a top-level layout was done to depict the systems that interact with the mass memory module, how they are connected and where the memory system fits in on a satellite, see figure 5.2. Three systems are shown to have access to the mass memory unit which is shown as a dark box. Figure 5.2 gives no information on the internal architecture of the mass memory system but shows the surrounding units and the way they interact with the memory.

Although the MPU will be designed to be a part of the mass memory unit, the overall architecture of the memory unit is not important to this study, since some of the goals of the MPU system was to be modular and independent from it. The mass memory controller will be modeled as a system which merely inputs and outputs data to the flash device as well as interfacing the commands to it. This is sufficient because any interface or controller would have to use the basic command and control structures dictated by the flash device's datasheet.

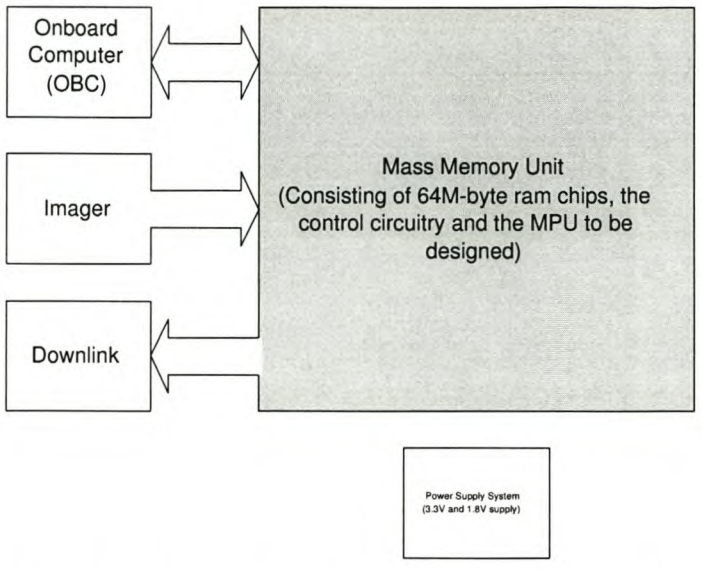


Figure 5.2: *A top-level layout of the relevant peripheral subsystems*

5.2.4 Concepts for MPU design structure

Now that the external factors to the mass memory system have been realised, the concept design of the MPU can begin. Figure 5.3 shows a proposed design structure for the MPU. This figure is an internal section of the Mass Memory Unit shown in figure 5.2.

The MPU encapsulates the control architecture without internally interacting with it. It therefore remains independent from the control. This gives the MPU the capability to be added to any control architecture interfacing flash memory which is one of the design goals.

The thoughts behind this structure was that the incoming data or outgoing data passes through the MPU layer. This allows the MPU to control what data reaches and exits the flash interface layer. Having control over the data flow allows the data to be processed, therefore making the EDAC implementable in the MPU layer. The external control signals to and from the peripheral components, such as the OBC, however, can be directly passed through the MPU layer therefore making the MPU transparent to the overall system which is another of the design goals.

Control over the power supply to the flash chip will also be required by the MPU and therefore the power rails must also pass through the MPU layer.

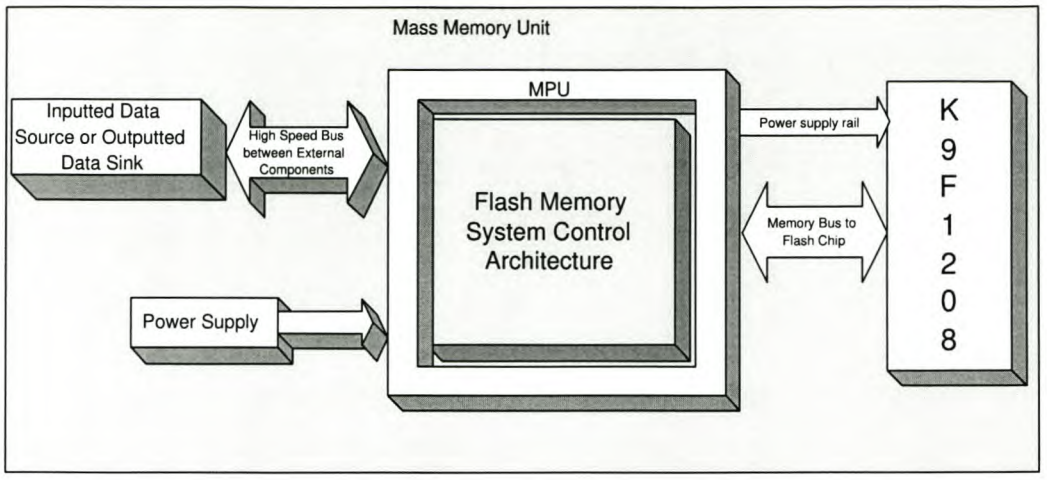


Figure 5.3: *Concept design of MPU module inside Mass Memory Unit*

In order to give the MPU, itself, structure, the various requirements of the study must be realised and assessed. Figure 5.4 is a flowdiagram which gives a description of the various error states that can occur, according to section 3.4, and suggests a mitigation scheme for each of them. Each error detection, correction and mitigation scheme that was presented in section 3.4 can clearly now be seen and development of each scheme can follow. The entire system design and development will be structured around this diagram since it depicts all the necessary areas of the study that need to be created and implemented.

The main subsections of the MPU are thus:

1. Error Detection and Correction (EDAC).
2. Single Event Functional Interrupt (SEFI) Handling.
3. Single Event Latchup (SEL) handling.
4. Bad Block Management.
5. Error reporting to OBC.

It can also be seen from figure 5.4 that certain separate error schemes are linked, such as determining whether a SEFI or SEL has occurred. Both schemes have to check the state of the current being drawn by the flash chip therefore will probably share hardware and software resources later in their respective developments.

The fact that in certain mitigation schemes the current process, that became faulty, has to be rerun means that there has to be a small amount of interaction between the MPU and

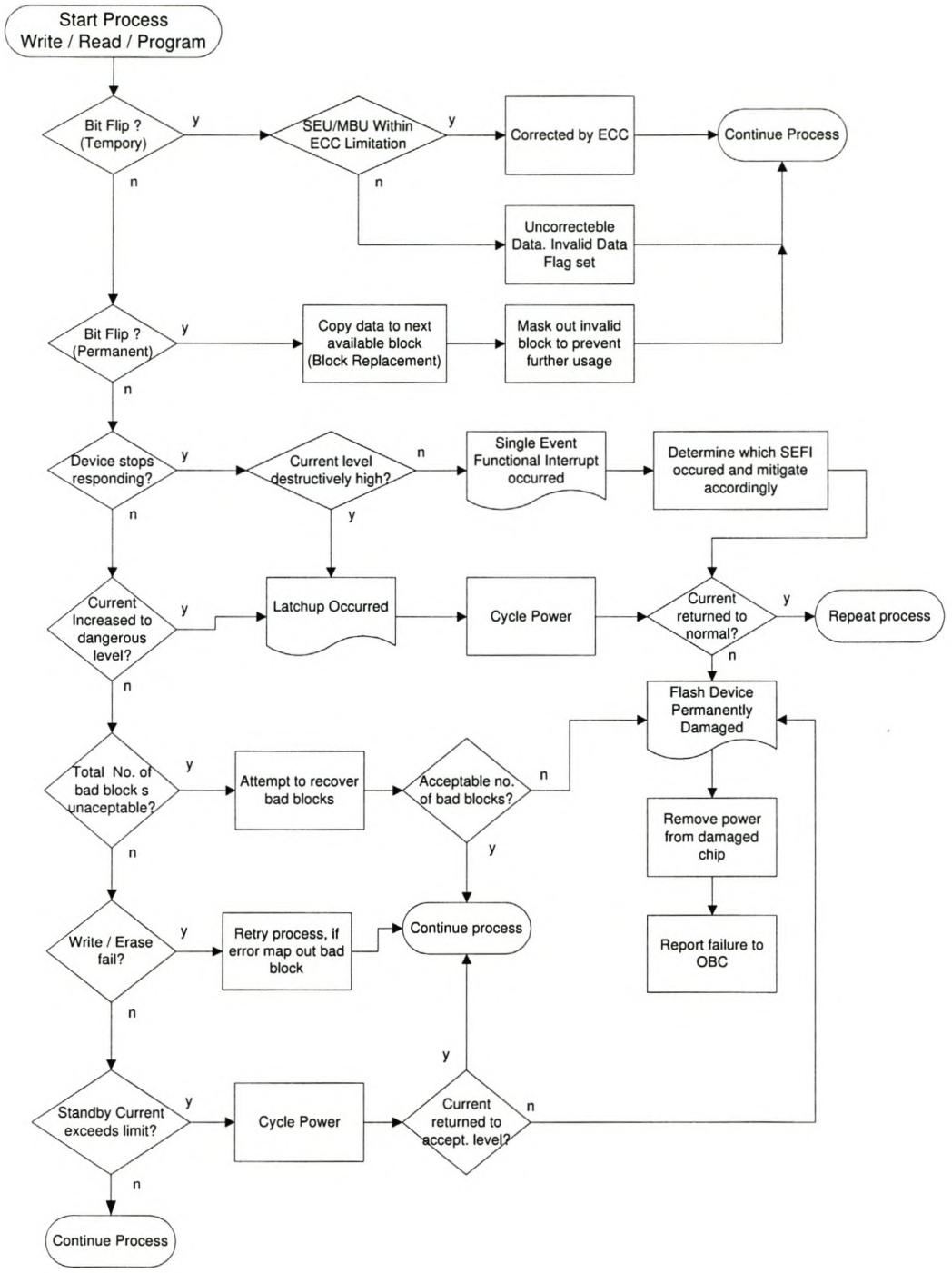


Figure 5.4: *Flowdiagram showing all possible error states and suggested mitigation schemes*

the Mass Memory controller in order for the latter to be informed of the error. The MPU is therefore not completely independent from the mass memory controller as previously believed. The interaction between the two, however, will be kept to a minimum, such as an error signal with no data transferred.

Concept designs of each of these five sections are completed in the rest of section 5.2.

5.2.5 EDAC system

As data arrives from the peripheral sources, such as the imager, it must be encoded before it arrives at the mass memory controller where it is sent to the flash memory hardware for storage. Similarly as data is requested from the memory controller it must be decoded and corrected before being transmitted to the intended recipient.

The EDAC system must therefore intercept the incoming data before it arrives at the mass memory controller. Since one of the goals was transparency of the EDAC to the memory controller, the EDAC system therefore cannot have any control of the data arrival or data retrieval from the external peripheral components. The EDAC system must therefore be designed to be in a ready mode, waiting for data to arrive from either side, the data to be encoded and stored or retrieved and decoded.

As external data to be stored arrives it must immediately be passed to the encoder which in turn transmits the encoded data to the memory controller which stores it in the flash memory. Similarly as encoded data arrives from the memory controller it must be sent to the decoder which must perform any needed error correction and then output the decoded data to the external recipient. This will ensure that the EDAC encoding, or decoding, occurs as fast as possible to ensure the data rate is maintained. A backlog of data at the encoder which cannot be maintained would cause the data to be corrupted or lost, and therefore superfluous, since the data that arrives cannot be retransmitted (an image is taken while scanning a section of the earth as the satellite rotates around the orbit, therefore cannot be retaken immediately). A backlog at the decoder would be less critical but extremely difficult to correct and therefore should be avoided at all costs. The encoding process, as mentioned in earlier chapters, must occur on-the-fly and at a suitable speed to ensure no unmaintainable backlog occurs.

In order to design for these requirements, the chosen error correction code, Reed Solomon code scheme (see section 4.3), must first be examined. This will reveal the complications or challenges that must be overcome in order to make the EDAC system successful.

Reed-Solomon Concepts

As discussed in section 4.2 a Reed-Solomon code is quite complex and requires a fairly large amount of computing power to be implemented. The design of the actual low-level encoding and decoding process is beyond the scope of this thesis, however can still be implemented. A specific Reed-Solomon encoder/decoder integrated circuit can be purchased and implemented into the design. Xilinx offer a Reed-Solomon ASIC that can perform the necessary data encoding and decoding, however, to employ an entire hardware component just for a single process, with no other possible usage, would be deemed overkill and therefore another solution should be sought. A more viable option would be to implement the Reed-Solomon encoder/decoder in an FPGA. This allows the hardware, the FPGA, to be used for other purposes besides error detection and correction.

The best way to implement a Reed-Solomon system is by installing a *Megafunction* in an ALTERA FPGA. *Megafunctions* create high-level entities, that can be altered to suit the required specification, which can be programmed in an FPGA. FPGA's are programmed using VHDL (very high speed integrated circuit hardware descriptive language). *Megafunctions* merely create 'ready made' VHDL entities for ALTERA FPGA's. ALTERA manufactures many different FPGA's which differ largely in size, capability, speed and cost. Determining which model of FPGA to choose will be done at the end of the low-level design phase when all the hardware requirements are known.

The control and operation information of the Altera Reed-Solomon component was obtained in [1]. The important factors of the encoder/decoder that dictate its operation and influence the system design are:

1. The encoder basically accepts symbols and outputs them on the next clock edge. This continues until the chosen amount of data symbols per code block have been accepted. The encoder then outputs the chosen amount of code symbols to complete the code block and can start a new code block by accepting new data symbols again, see diagram 5.5.

2. Similarly the decoder accepts the data symbols followed by the code symbols but unlike the encoder, only starts outputting data symbols once the entire code block has been read in. A *valid* flag is set when the outputted data is available and valid. This flag is cleared if the outputted data is invalid. There is also a waiting period from when the last block symbol is inputted until the first decoded symbol is outputted.

3. Encoding and Decoding can operate at speeds greater than 100MHz which is more than sufficient to meet the requirements.

4. The decoder has a bypass utility that allows the incoming symbols to pass through unaltered.

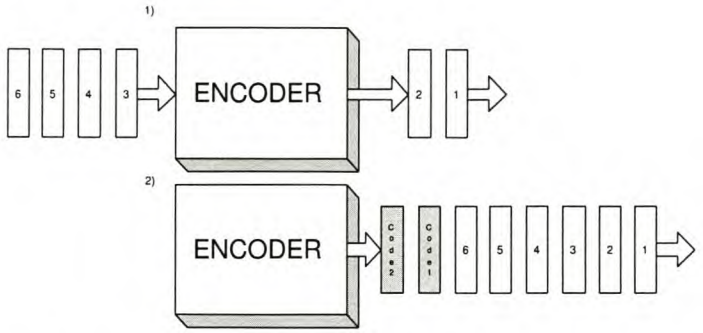


Figure 5.5: *Encoding showing (1) the data symbols being streamed through the encoder and (2) the code symbols created following the last data symbols*

The design

The previous subsection gave valuable insight into the operation of the chosen Reed-Solomon encoder and decoder, and what is required to meet the EDAC constraints. Both the encoder and decoder need controllers, or drivers, to manage their respective operations, eg: there would be necessary key control signals such as informing the encoder/decoder that a new code block is being started or that a new symbol has arrived, etc. A bypass utility for the encoder should also be implemented in case the encoder becomes damaged and starts corrupting data. One main control unit would be sufficient in controlling the operations of the two EDAC entities and bypass system.

The code symbol creation and insertion into the symbol stream done in the encoding process produces latencies into the system. In the decoding process, waiting for the first decoded symbol to be outputted after the last block symbol has been inputted into the decoder also creates latencies. These latency problems will cause data to be lost due to a backlog being formed and must therefore be mitigated.

A solution to this problem is to buffer the data on both sides of the encoder and decoder. The data will therefore be fed directly into a first-in-first-out (FIFO) queue and can easily be monitored by watching its *not_empty* signal which informs the EDAC controller when data is being queued. The FIFO queue will ensure that the order of the data stays intact, therefore no corruption can occur. On completion of encoding or decoding, the data can be sent to an output buffer similar to the previous FIFO queue. From there the data can be handled in whatever way the next recipient requests.

The size of the various FIFO's will however differ. For example, the input FIFO to the encoder only has to queue symbols while the code symbols are being outputted therefore will have a maximum depth of the number of code symbols created. The input FIFO to the decoder, however, can end up queueing up to an entire codeword since the decoder has to input a whole codeword before outputting a single symbol. All this buffering requires a fair amount of memory (An entire codeword can be up to 255 bytes). A solution would be to use internal RAMBLOCKS that are embedded in certain FPGA's. Luckily enough the same make of FPGA, ALTERA, that was chosen for the Reed-Solomon coding implementation also manufactures FPGA's with internal RAMBLOCKS. Therefore the same hardware can perform both needed tasks.

Another vital aspect is the reporting of errors that could occur in the EDAC system. They are:

1. The output buffers of either the encoder or decoder could become full due to the next stage recipients not emptying them.

2. Too many errors occurring in a codeword would cause the decoder to fail and that data block would become corrupted.

3. Not enough data symbols could arrive at the encoder to fill the last data block. Therefore not getting encoded which could leave that data vulnerable to any number of bit upsets

Error number 3 can be overcome by setting a watchdog timer checking the arrival rate of the data symbols. If the watchdog times out then it can be assumed that the dataflow has ended and the rest of the final block can be filled by arbitrary data.
The other 2 errors must be reported to a higher level controller which can then handle the errors and, if needed, report to the OBC.

An overview of the high-level design concept for the EDAC system is shown in figure 5.6.

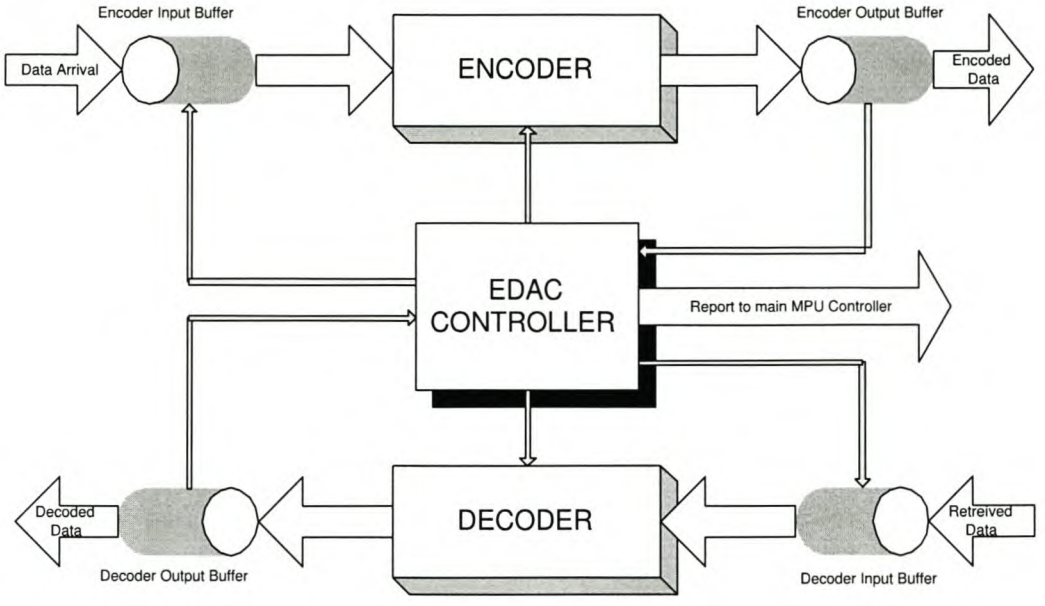


Figure 5.6: *High-level concept design for EDAC subsystem*

The four FIFO buffers and the respective dataflow directions can clearly be seen. It is important to notice that the EDAC controller only has control over the two input buffers but not of the output buffers. The two output buffers only report whether or not they have become full to the EDAC controller. Their respective controllers are based in their respective recipient stages.
The EDAC controller reports to the main MPU controller. This was decided upon to keep the subsystems modular and therefore requires a main top level controller which monitors and controls all subsystems. The main controller will be designed at a later stage.

This completes the concept design of the EDAC system. The actual low-level design and implementation was done in 6.3.1.

5.2.6 SEFI handling

The complex nature and behavior of SEFI's was discussed in detail in subsection 3.4.2. It was concluded that SEFI's could not be prevented therefore the only mitigation scheme possible was to observe the failure of the flash device and then correct that failure. Correcting SEFI's according to subsection 3.4.2 entails cycling power, or resetting the device, or restarting the process that the flash was executing when failure occurred. But in order to choose a mitigation scheme certain information about the flash must be known such as:

1. What type of operation was the flash executing at the time of failure?
2. How much electrical current was the flash device drawing during this operation?
3. Were the previous operations, before the device failed, successful?

Most importantly, before any mitigation can take place, the occurrence of a SEFI must be detected. Using all these facts the SEFI system concept can be designed.

The design

By looking at the required mitigation and monitoring schemes highlighted in subsection 3.4.2, certain devices were required namely:

1. A **current monitoring** system to report how much current is being drawn by the flash device.
2. **A power control circuit** which can control the current supply to the flash device therefore making power cycling possible and even power removal if necessary.
3. A **watchdog circuit** which can determine if a SEFI has occurred by monitoring its operations and timing out when the flash device stops responding.
4. A **reset circuit** which sends the reset command to the flash device. This command causes the flash to abort any operation in execution, clearing the command register and status register.
5. A **flash operation monitoring** system to monitor what the flash device is doing at all times.

Similar to the EDAC system, a SEFI controller was decided upon in order to monitor the 5 SEFI subsystems just discussed. The SEFI controller controls and monitors each of the subsystems and reports any errors to the main MPU controller just like the EDAC controller. The discussion of each of the other devices follows.

The **current monitoring** system must be connected to the power supply rail to the flash chip in order to analyze the current being drawn by the device. This will require the monitoring system to be inserted in series with the supply rail. A current sensor is not implementable using an FPGA since FPGA's have digital logic input/output pins not analog, which is required. Therefore an external current sensor must be chosen.
Most current sensors operate by inserting a small resistor (< 1 ohm) in series with the current line that is being measured. The tiny voltage that spans across the small resistor is then amplified to give a current measurement. This current measurement is still an

analog signal and must be converted to a digital signal in order for an FPGA to analyze. Analog to digital (A2D) converters perform this task and will therefore be added to the current monitoring system. The current sensor measurement will therefore be sent to the A2D converter which in turn will output the digital equivalent value to the current controller implemented inside the FPGA where the current measurement can be analyzed. The current controller also has to control the operation of the A2D converter, such as its sampling rate.
A suitable A2D converter and current sensor will be chosen in the low-level design phase. The main factor of the converter is its resolution. A greater resolution gives a higher degree of accuracy but will obviously cost more. The radiation hardness of the current sensor and A2D converter is also a major factor influencing the device choice.

The **power control** circuit must be able to turn the power supplied to the flash device on and off with relative high speed. Similar to the current measurement system, the power control circuit must also be in series with the power supply rail to the flash device. The flash chip must be supplied with 3.3V when 'on' and 0V when 'off'. According to [14], the supply voltage can vary from 2.7V to 3.6V. This allows some leeway in the control circuit which will be developed in the low-level design phase. The power circuit will be controlled by the SEFI controller implemented inside the FPGA. Therefore the switching device must be compatible with a digital input/output pin from an FPGA.

The main aim of the **watchdog circuit** is to monitor the flash device's operation and timeout when the flash device suspends. The non-operation of the flash device cannot be detected by checking its status byte therefore requiring the use of a watchdog circuit. In order to monitor the flash device's operation the bus lines connecting the flash must be monitored. One of the goals of the MPU is to remain independent from the Mass Memory Controller therefore interfacing with it would not be acceptable. This leaves the latter as the chosen option. By monitoring the physical bus lines that connect to the flash, all operations and data being sent or received with the flash chip can be obtained, without any knowledge of the Mass Memory Controller/interface. This ensures the independence of the MPU shown in figure 5.3.
A great option would be to send the bus lines to the **flash operation monitoring** system which can then analyze what operation the flash is executing. Therefore as a new command arrives at the flash device, the flash operation monitor can determine what type of command it is and in turn inform the watchdog circuit, which can then monitor the operation accordingly. It is imperative for the watchdog controller to be informed as to the type of operation being executed by the flash because different operations react in different ways.

The actual watchdog circuit can either be implemented in an FPGA or by using an external circuit. In order to minimize extra hardware it will be implemented in an FPGA, which is already required by previous designed devices, along with the watchdog controller. Similarly the flash operation monitoring system will also be implemented in an FPGA.

The **reset circuit** poses a problem because, unlike the previous SEFI systems, the reset circuit has to actively interact with the flash device. This is because the reset command has to be sent to the flash chip which would entail taking control of the physical bus lines controlling the flash. In order to take control of these lines would require hardware multiplexers and control lines. This extra hardware is unwanted and not a suitable solution. As mentioned in section 5.2.4 a small amount of interaction between the MPU and mass memory controller is inevitable. A viable solution would be a single reset line between the MPU and mass memory controller which informs the latter of the SEFI and initiates a reset command to be sent to the flash device.

Now that all the required systems are in place the SEFI controller, previously mentioned, has all the information and control devices to perform the necessary mitigation strategies. The SEFI controller, as with the EDAC controller, will report all necessary activities to the main MPU controller which, if need be, can report to the OBC. An overview of the high-level design concept for the SEFI system is shown in figure 5.7.

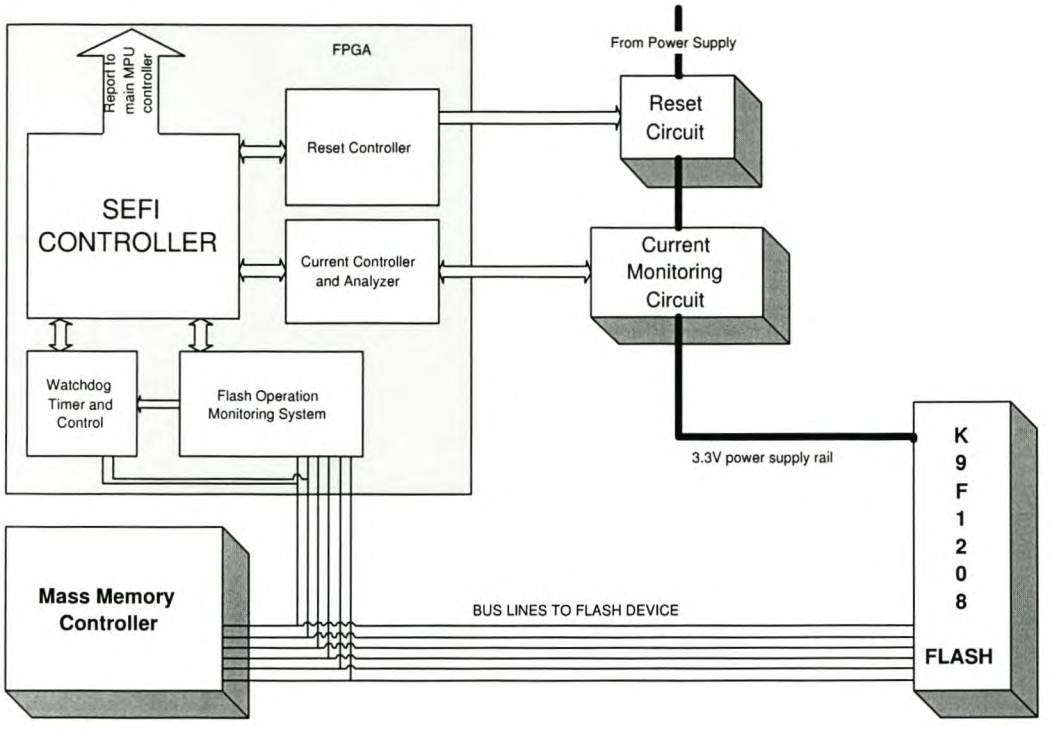


Figure 5.7: *High-level concept design for SEFI subsystem*

Figure 5.7 clearly shows all the subsystem elements inside the FPGA (the grey block) as well as the necessary added hardware. It is unknown how the mass memory controller/interface is implemented but if it is implemented within an FPGA then the SEFI system could also be implemented in the same FPGA, therefore sparing the need for extra hard bus lines to link the original bus, as seen in the figure 5.7.

5.2.7 SEL handling

Single event latchup is a hard error which can cause permanent damage to the flash memory. The cause of latchup was investigated in chapter 2.3.3, and the resulting effects caused in flash memories in chapter 3.4.3. Whether or not the current flash memory process fails due to the latchup, the latchup state must be mitigated immediately in order to protect the device from physical damage.

Firstly latchup must be detected. The best way to monitor latchup is to monitor the current being drawn by the flash device. Therefore the current supply rail must be monitored at all times. However, a current monitoring system has already been conceptually designed in the previous section, section 5.2.6. This monitoring system can be used to detect latchup as well as SEFI's. So no extra hardware will be needed to detect latchup.

However, in order to detect latchup, much higher currents than originally planned for the SEFI current system must be monitored. The current monitoring system must therefore be calibrated to handle a wider range of current values, which will impact on the A2D converter choice in order to maintain the intended resolution capability.

Mitigating latchup entails removing power from the device and then re-applying power. The high current levels should then return to acceptable levels. If the current does not return to an acceptable level then the device can be acknowledged as being permanently damaged, therefore power should be completely removed from the device and its malfunction reported to the master MPU controller which must then report to the OBC. The power to the flash chip must be controllable in order to remove and re-apply power. As with the current monitoring system, a power control system was already conceptually designed in the previous section and will therefore not require re-designing. The entire SEL system can therefore be implemented by enhancing the SEFI system to meet the SEL requirements as well as its own.

5.2.8 Bad Block Management

Bad blocks, or invalid blocks, were discussed in section 3.4.4. This involved their definition, preliminary facts about their occurrence and reasons for the requirement of a mitigation scheme. It was realised that a bad block does not affect the surrounding blocks and can be mitigated by simply not *using* the bad block in any way. An effective measure to remove a bad block from usage is to map it out of the addressing scheme. The address handler will no longer address the block therefore mitigating all the problems associated with that block. A system must now be developed in order to detect and manage the occurrence of bad blocks. The main goals of the system are:

1. To detect and record the occurrence of new bad blocks.
2. To store the position of all known bad blocks using as little resources as possible.
3. Quick simple access to the bad block data.
4. Knowledge of the total number of bad blocks.

The design

Preliminary discussion involving the bad block system revealed that the reading/ programming/ erasing, and therefore addressing, of data to the flash chip will be done by the memory controller. As mentioned in the previous section, all interaction with the memory controller is kept to a bare minimum and therefore the addressing scheme used

by the controller will be unknown to the MPU. A small amount of interaction between the two is necessary in order to convey the bad block information to the memory controller. Therefore the bad block management system must provide a small simple interface to the controller so that information can be transmitted between the two. How the memory controller uses the bad block information is inconsequential, the bad block system must just be able to provide the required information.

Firstly an appropriate storage structure and scheme must be developed. This can dictate what hardware is required and then the appropriate implementation can follow.
A solution is to record the address value of each of the bad blocks. The addresses would then be stored in a look-up table which can be externally accessed by the memory controller in order to perform bad block checking. This would entail storing the entire block address which would require roughly 2 bytes per bad block address since a block address, according to figure 5.1 (bits A14 to A25), is 12bits wide. This bad block scheme was found unsuitable because:

1. With only a few bad blocks this form of monitoring would be efficient, but as the number of bad blocks increase, the lookup table would expand and become extremely large. For example, if half the flash device's blocks became invalid then the amount of storage space required by the bad block table would be 4,096 bytes.

2. The amount of memory allocated to implement the lookup table would be unknown and hard to choose, which could become critical if multiple flash chips were to be protected requiring multiple lookup tables.

3. A complicated addressing scheme within the table would also be required in order to arrange the addresses accordingly as they are added to the table, and to keep track of what addresses are stored.

4. Lookup time would also be variable and increase as the table size increases. For addressing purposes, the variation in time to check the bad block table is unacceptable.

Due to the above reasons a better alternative was investigated. Instead of storing addresses it was chosen to represent the entire flash memory block address space as a bitmap. The bitmap being a table of fixed size, in which each block of flash memory is represented by a single bit, a *flag bit*. A flag bit can either be set as valid, representing a valid block, or invalid, representing an invalid block. The flag bits are numbered and stored in order with their respective flash memory block number, see figure 5.8. Note block 35 being marked as an invalid block. The advantages of this bad block scheme are:

1. The exact memory size required by the bad block table is fixed unlike the lookup table. This allows the designer to make decisions relating to its storage and operation. According to subsection 5.2.2, the flash chip concerned has 4,096 blocks. The bad block table will therefore be only 4,096 bits in size. Compared to the lookup table scheme, a lookup table of equal size would only allow the storage of 256 block addresses, which is only 6.25% of the total block addresses. The size advantage can clearly be seen.

2. The exact location of each block in the table is known therefore checking its validity is instant, requiring no lookup time as with the previous scheme. The only time delay will be in interfacing with the bad block table which will be kept to a minimum.

3. The addressing scheme for the bad block table will simply be the number of the requested block from the flash memory. Therefore intricate addressing is not necessary which simplifies the design considerably.

4. An increase of bad blocks does not affect the table size nor does it affect the speed of the bad block table. This is a major advantage over the previous scheme.

ADDR	+0	+1	+2	+3	+4	+5	+6	+7
0	v	v	v	v	v	v	v	v
8	v	v	v	v	v	v	v	v
16	v	v	v	v	v	v	v	v
24	v	v	v	v	v	v	v	v
32	v	v	v	i	v	v	v	v
40	v	v	v	v	v	v	v	v
48	v	v	v	v	v	v	v	v
56	v	v	v	v	v	v	v	v
64	v	v	v	v	v	v	v	v
72	v	v	v	v	v	v	v	v
80	v	v	v	v	v	v	v	v

Figure 5.8: *Concept design of bad block table structure*

The table should be initialized using the manufacturer's information mentioned in subsection 3.4.4 and shown by figure 3.5 so that initial bad blocks will be mapped out immediately. As new bad blocks occur and are detected the table should be updated accordingly. This will be a simple process of accessing the respective flag bit and marking it as *invalid*. Detecting new bad blocks was discussed in subsection 3.4.4 which basically involved checking the flash memory's status register. In order to achieve this a bad block controller must be implemented which monitors the status byte returned by each operation. If an error occurs in an operation then the controller must perform the necessary updating of the bad block table. The bad block controller must therefore monitor:

1. What address was last used by the flash memory.
2. The status byte returned after each operation.

This can be achieved by monitoring the control and data lines connected to the flash chip. However, a monitoring system on these lines has already been implemented by the SEFI subsystem designed in section 5.2.6. Therefore the bad block controller can be implemented in that same FPGA without requiring extra hardware to be added. Within the controller, an address decoder can be implemented which watches the control lines for an address to be applied. When this occurs the address is obtained and stored while waiting for the status of the operation to be returned. A status decoder can also be implemented within the controller to watch for the status byte. If the returned status byte shows a successful operation then the address can be discarded otherwise if unsuccessful then it can be used to update the bad block table.
As with the previous sections, the operations of the bad block system must be reported to the main MPU controller which in turn will inform the OBC. The bad block controller will be responsible for reporting to the MPU controller.

Now that the bad block scheme has been chosen it must be implemented. The entire contents of the bad block table needs to be stored in a non-volatile memory so that when power is removed the table is not lost. The table also needs to support simultaneous random access, reading and writing, so that any contents of the table can be accessed as well as updated at the same time. The possible solutions are:

1. **Store the table in the flash itself**. This is an option that will satisfy the non-volatile requirement. But the table requires random access which is not feasible in NAND flash memory, it would be too time consuming and would require control over the flash which is not possible due to the independence of the MPU and memory controller. Updating the table would also be difficult in that flash memory is first required to be erased, therefore temporary storage of the table will be necessary. The bad block table checking would be extremely slow since the flash will be occupied at the same time with the running program. All the above reasons make this an unattractive option.

2. **Store the table in separate non-volatile memory**. This is similar to the previous option but with the table being stored in different non-volatile memory. The non-volatile requirement is satisfied as well as the random access requirement, if a suitable type of memory is chosen such as EEPROM. The bad block operations can be run in parallel with the flash memory operation which will satisfy all timing requirements. Even though the extra memory required to store the bad block table is relatively small, extra hardware will still be required to be implemented. This

is an unattractive option since extra hardware is not sought after. Extra hardware means extra power, space and possible errors induced by radiation.

3. **Store the table in flash and temporarily in an FPGA**. Similar to option 1, the table is stored in the flash memory but before the memory is used, the bad block table is loaded into RAM where it can be accessed quickly and in parallel with the flash operations. The table in the RAM can also be updated during normal operation which is a requirement. Then at the end of flash operations the bad block table in the RAM can be stored back into the flash if altered. Since flash blocks can become invalid, the bad block table can be stored in multiple locations to reduce chances of it being lost. The RAM required to contain the table can easily be implemented in a RAMBLOCK within certain FPGA's. The FPGA chosen to implement the EDAC system in has the requirements suitable for the RAMBLOCK and can therefore be used. This is an attractive feature since no extra hardware will be needed to implement the bad block system. The only problem will be if power is lost between bad block storage then any new bad block locations will be lost. But their occurrence should be detected on the next operation and so should not prove to be a major problem. Block '0', the first block, on the chosen flash chip is guaranteed to be valid and is therefore a suitable choice as the primary position of the table.

Option 3 is by far the most suitable choice and will be implemented in the low-level design phase. However, this will require the mass memory controller to retrieve and store the bad block table at the correct location, which again requires a small amount of interaction between the MPU and mass memory controller. A concept design of the bad block system is shown in figure 5.9.

5.2.9 Error Reporting

In the previous sections, each subsystem controller reported to a master MPU controller which in turn communicated with the OBC. The subsystem reporting is all internal to the FPGA and therefore can be achieved by simply linking the various controllers with internal signals. This does not require external hardware or a communication protocol to be developed. However, the master MPU controller must communicate with the OBC which requires external hardware and a communications system to be installed. Since no exact specification was outlined for communication with the OBC, all viable solutions will be investigated and the most suitable solution chosen.Firstly the factors influencing the decision must be outlined.

As errors occur in flash they must be reported to the OBC so that the system is aware of

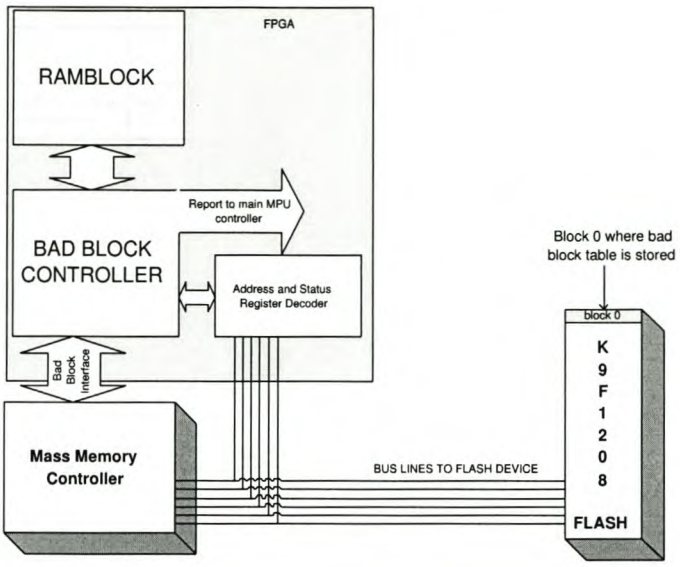


Figure 5.9: *High level concept design for bad block system*

what is happening in the mass memory system. Reports include all major errors such as new bad blocks, SEFI's, SEL and EDAC errors. The OBC will then know what state the mass memory system is in and can therefore make adjustments, or inform the engineers accordingly. Error reporting must be independent from the working of the memory and memory controller so that if there is a failure or power reset then the errors can still be reported. The speed of the error reporting system is quite critical. Influences on the required speed will be EDAC failure and error detection reporting, bad block occurrence reporting, SEFI and SEL reporting. For example, while data is being decoded an EDAC failure would cause all the current data to be corrupted and therefore must be reported immediately so that the memory system can compensate for the error accordingly, such as requesting a retransmission of the data. Since the data rate is roughly 10 M-bytes per second, the reporting rate must be similar in speed. The reported data must contain information such as:

1. Type of failure.
2. Position of failure in memory (bad block).
3. Nature of failure.

So for a single error, a number of bytes of information has to be sent to the OBC. Possibilities for the communication system are:

Serial Communication (RS232)

Serial communication is fairly simple to implement using 3 physical lines. The bit rate is determined by the clock rate. Each bit is sent separately on the line in blocks of 10 bits, 8 data bits + 1 start bit + 1 stop bit). The overhead is therefore 20%. Serial comm reaches speeds of 110KHz baud rate. Compared with the 10M-Byte per second transfer speed, this comm. is rather slow and could result in major bottlenecks. Implementing a serial comm. system requires a serial transceiver which is an extra piece of hardware. Only 1 transceiver is necessary so therefore it is not such a burden to the system. The pro's of using a serial comm. is that it is easy to implement, requiring an interface which can be implement in an FPGA and a RS232 hardware transceiver. The downside of serial comm. is its slow speed.

Controller Area Network Bus Communication (CAN Bus)

A controller area network bus (CAN bus) will allow the error reporting system to be connected to a network at very high speeds. These controller area networks are common on systems such as satellites where a variety of different hardware types have to communicate. A CAN bus can reach speeds of 1M-byte per second which will be sufficient to meet the timing requirements. CAN bus comm. requires 3 physical lines, a CAN bus controller and a CAN bus transceiver. The controller can be implemented in an FPGA which then sends and receives data via the transceiver which is separate hardware. Implementing a CAN bus seems similar to a serial comm. but has certain differences: the controller is much more complicated and therefore requires more FPGA resources. The CAN bus transceiver is also far more expensive than the serial counterpart which is also a major influence. The CAN bus therefore has a higher speed but is far more complex and expensive to implement.

I^2C Communication

I^2C is a bi-directional 2-wire bus communication for efficient inter-IC control, [13]. Only two wires are required , a serial data line (SDA) and a serial clock line (SCL). Each device on an I^2C line has a unique address and has a master / slave relationship. The speed of this comm. ranges from 400K-Bits per second in fast mode to 3.4M-bits per second in high speed mode. Implementing I^2C comm. requires the 2 bus lines and a controller. Unlike the previous two forms of communication, I^2C does not require a transceiver. The linked IC's however must be I^2C compatible. Since I^2C can run straight off the FPGA no extra hardware besides the 2 comm. lines is required which is an attractive feature. The controller is fairly complex but can be implemented within the FPGA. I^2C communication is therefore fast enough to meet the requirements and is fairly cheap and simple to implement.

Decision and Design

In order to decide the most suitable solution a comparison was done between the three, see table 5.1, where each deciding factor is ranked from 1 to 3, 1 being the best and 3 being the worst. The low cost of I^2C is contributed to the fact that no hardware transceiver

Table 5.1: *Comparison of Communication Architectures*

	Cost	**Speed**	**Implementation**
Serial	2	3	1
CAN	3	1	3
I^2C	1	2	2

is required unlike the other two. Although I^2C can achieve a faster speed than the CAN bus, its average speed is slower and therefore was marked second in speed behind the CAN bus in the table.

The totals for each solution are: Serial = 6, CAN = 7 and I^2C = 5. I^2C is therefore the most beneficial implementation. However, it was chosen to rather implement serial communication for the following reason: The communication protocol required by the OBC is unspecified and merely a conceptual design feature in this study, therefore choosing one to implement is trivial and should not require lots of resources and time. In order to implement an I^2C interface in a personal computer, to test this system, would be difficult and time consuming. Serial communication can easily be used in a personal computer using the serial port and a serial data capture program. The RS232 transceiver required is also fairly cheap and readily available, therefore not a cost increase over I^2C. So therefore serial communication proved a better option than I^2C for testing putposes. The concept design is shown in figure 5.10.

The peripheral subsystem controllers and their reporting to the master MPU controller can clearly be seen. The error report control is handled within the MPU controller which in turn sends the error messages to the serial interface / controller. Here the messages are converted to the specific protocol and sent on the serial bus to the OBC. The serial bus can transmit and receive data to and from the OBC by implementing two separate serial lines. The MPU therefore does not necessarily have to be fully functional all the time. It can be in a low-power-standby mode until informed by the OBC that data will be arriving shortly. The necessary subsystems can therefore be brought out of standby mode just for its required time.

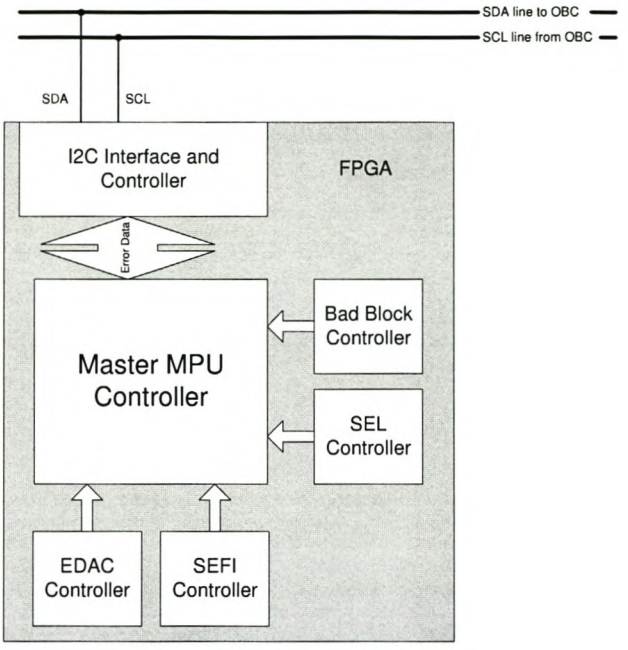


Figure 5.10: *Concept design of communication system structure*

5.2.10 Overall Concept Design

By combining and simplifying the previous concept designs, from 5.2.5 to 5.2.9, an overall concept design was developed, see figure 5.11. All the subsystems and their relative positions within the design can be seen. This gives the designer a perspective from which to begin and plan the low-level design.

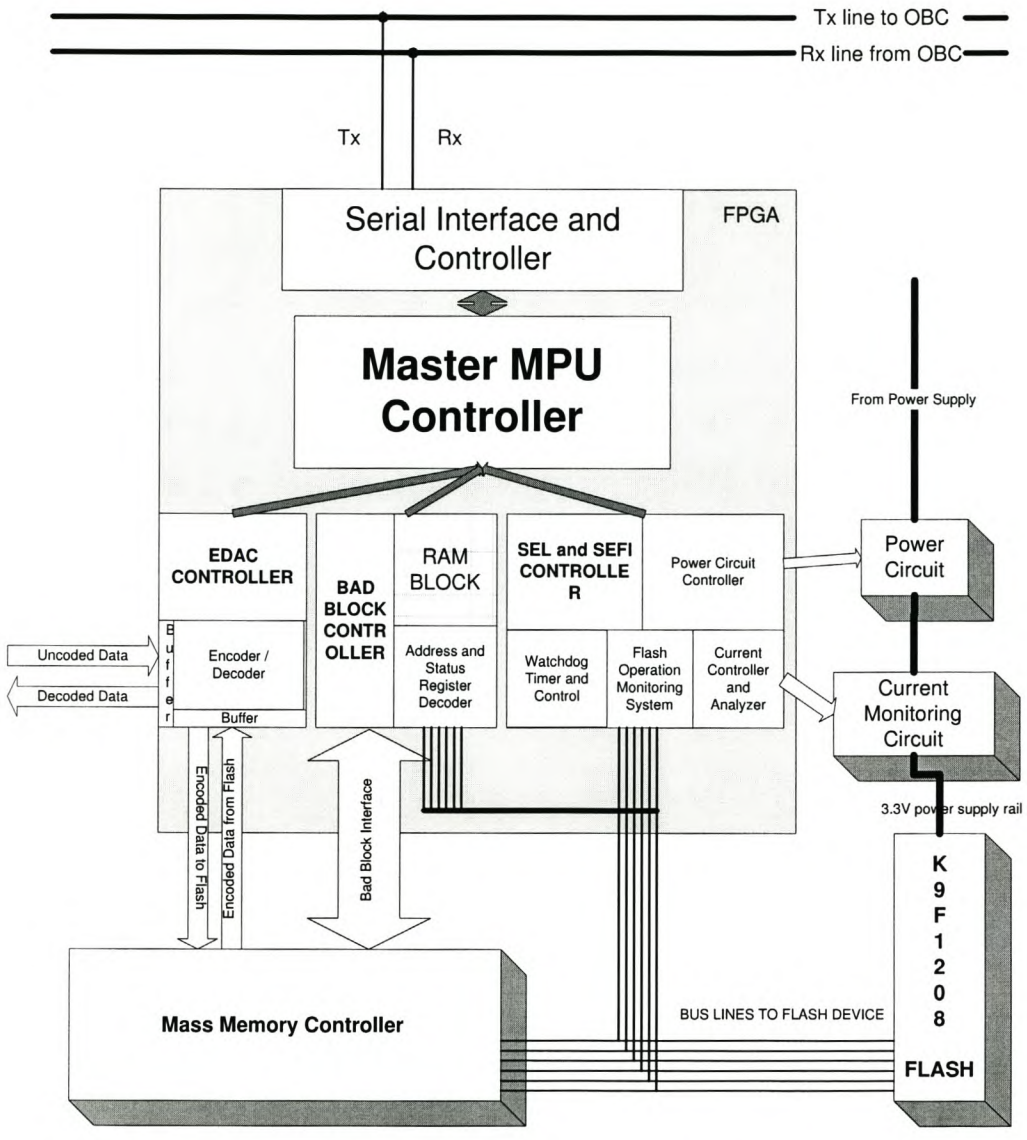


Figure 5.11: *Overall concept design of MPU system*

Chapter 6

Low Level Design of MPU System

So far, the detailed concept design was completed in chapter 5. There, each of the requirements for the respective subsystems were analyzed and discussed, forming functional blocks which had to be implemented. The next step was to realize these blocks using the necessary hardware and software. The low-level design allowed the design to be explored on a chip level. This involved component selection and reliability discussions.

This chapter begins with a brief description on how hardware designs were implemented in a FPGA and simulated using specific software. The next subsections detail how the major implementations in each design section were achieved. Design sections being: the EDAC system, the bad block system, the SEL and SEFI system and the error reporting system. The idea was not to bore the reader with the intricate design of each system but to highlight and discuss the main concepts and difficulties overcome in implementing each system. This chapter ends by evaluating the various subsystems by using simulation software.

6.1 FPGA and Programming Basics

As decided upon in chapter 5, the bulk of the low-level design was to be implemented in a FPGA. This required a suitable FPGA to be chosen and the respective programming language to be mastered and implemented. Firstly a brief background to FPGA operation is given.

Basically a FPGA is a hardware device which can be programmed to perform an infinite number of tasks. It consists mostly of a large number of logic-elements which are configured upon being programmed by the hardware language. The hardware language which programmes the FPGA is called VHDL (very high speed integrated circuit hardware descriptive language). The written VHDL code must first be compiled, which is done

by the computer program associated with the chosen FPGA. The compiled code is then programmed into the FPGA using a separate hardware programmer. Once programmed, the FPGA will function as the VHDL code instructed.

In order to begin low-level design, a FPGA first had to be chosen so that the required programming software could be installed and then used. Choosing a suitable FPGA involved selecting one with the required features such as:

1. The FPGA must obviously be large enough. This involves having enough input / output pins which are needed in transferring data to and from the FPGA. Also the design must fit into the FPGA, therefore must have enough logical-elements to implement all the required design logic.

2. The speed of the FPGA is also a major factor. Most manufacturers offer similar sized FPGA's but with different speed grades, the faster obviously being more expensive. This leads to the next feature.

3. The FPGA must also be within the allowed cost budget.

4. The FPGA must provide all necessary internal components such as RAMBLOCKS and phase-lock-loops.

Using the above information, the selection was narrowed down to selecting an ALTERA CYCLONE FPGA. The new CYCLONE group of FPGA's from ALTERA have all the features that were required, such as *megafunction* support and internal RAM, and was one of the cheapest available FPGA's. The specific model could not be chosen yet until the final design was compiled so that the total number of logical elements required could be known. However, an important aspect of the low-level design is the frequency of the system clock supplied to the FPGA. Considering the maximum speed that the flash memory can operate as well as other hardware, such as the analog to digital converter, a system clock speed of **50MHz** was chosen.

The major subsystems of the MPU were described with VHDL. Throughout the low-level design phase the VHDL code was compiled and simulated using the ALTERA software program: *QUARTUS II*. This is a software development platform available to the students from the University of Stellenbosch. *QUARTUS II* was also used to program the FPGA via a *BYTEBLASTER II* hardware programmer which connects the FPGA to the parallel port of the computer. *Quartus II* was chosen ahead of the popular *Max+Plus II* design platform because the latter was not compatible with the new CYCLONE FPGA's.

6.2 Design Structure

It was important that a structured design approach was followed so that the design could easily be tested, maintained or expanded upon by other people. Structured design also promotes reliability which is important for a system that will be used onboard a satellite. Debugging is also made easier with a structured design set since the problem can be located easily and efficiently.

The design of the various VHDL blocks was state machine based. An attractive feature of state machine based systems is that extra functionality can be added to the design without any major changes to the design structure. Typically only a few extra states need to be added to perform the required function.

An important lesson was learned while programming which was to keep the design synchronous where possible. This means most assignments are performed on the rising edge of the system clock and not on the edge of other signal transitions. Another hard lesson learnt was to synchronize incoming signals from other hardware. These techniques promote stability and mitigate the effect of glitches.

6.3 Detail Design

6.3.1 EDAC design

The concept design for the EDAC system was constructed in section 5.2.5 and displayed in figure 5.6. From the design, three main areas were discussed and would now have to be implemented, namely: The *input / output buffers*, the *EDAC controller*, and the *Reed Solomon encoder and decoder*. The following subsections provide a detailed design of each of these main areas.

Reed-Solomon Encoder and Decoder

The Reed-Solomon encoder and decoder was implemented by using a Reed-Solomon *MegaCore* function from ALTERA. The *MegaCore* creates an encoder or decoder which can be altered to meet the required specifications concerning the Reed-Solomon coding scheme. The encoder and decoder are separate entities but must both be designed to accept and operate with the exact same Reed-Solomon scheme.

Choosing appropriate scheme Since the incoming data is 8 bits wide it made sense to set the symbol width to correspond with that width. Therefore, from section 4.2.3, the maximum codeword size is 255 symbols. This is a convenient choice for the codeword size since the page size for the flash memory is 512+16 bytes, therefore 2 codewords would fit perfectly onto a single page. The level of redundancy would now have to be decided. The maximum number of check symbols allowed by the *MegaCore* feature is 50 symbols. Choosing 50 check symbols results in a redundancy of 19% which gives a correctable rate, explained in 4.2.3, of 12.2%. This coding scheme is not acceptable due to the following factors:

1. The correction rate of 12.2% is lower than Hamming code correction rate of 12.5%. This is not acceptable even though Reed-Solomon allows multiple bit correction.

2. Implementing the (255,205) Reed-Solomon scheme in an FPGA requires over 5000 logic elements. This is a massive amount of logic and would require a much larger FPGA than expected.

3. If an error occurred within a large codeword, all the data in that codeword would be corrupted, therefore by decreasing the codeword size the relative vulnerability of data would also decrease.

The number of logic elements required to implement the function is linearly dependent on both the field size and the number of check symbols. Therefore by decreasing the codeword size, a better Reed-Solomon scheme could be found.

By choosing a codeword size of 64 symbols, or bytes, with 21 check symbols, the hardware resources required would greatly be decreased. The codeword size was chosen to fit exactly into a page of flash memory, therefore 8 codewords will fit per page. This (64,43) Reed-Solomon scheme requires <3000 logic elements which is a great improvement. The redundancy is increased to an acceptable 32% and gives an error correction rate of 24.4%. Due to these positive factors the **RS(64,43)** code scheme was chosen and implemented.

Data Buffers

The need for buffering was discussed in 5.2.5. It was concluded that FIFO buffers were suitable to perform the necessary queuing task. The depth and control of the respective buffers must now be decided upon and then implemented. A major factor influencing the buffer depth choice is the frequency of the system clock. A clock speed of 50MHz was chosen in the previous section.

Encoder Input buffer The purpose of this buffer was to queue bytes arriving at the encoder while the code symbols are being outputted by the encoder into the data stream. The depth required by the buffer can be calculated as follows:

1. The 50MHz system clock gives a period of 20ns.
2. The fastest time that is required to generate 21 check symbols and insert them into the data stream is therefore: $21 \times 20ns = 420ns$.
3. A new byte arrives every 100ns at the encoder, therefore 420ns/100ns=4.2 bytes will need to be buffered. By including a safety margin, a buffer of 8 bytes wide should be sufficient to meet the buffering requirements.

Encoder Output Buffer In the concept design phase it was decided to buffer the encoded bytes between the encoder and the memory controller. The memory controller can therefore retrieve the data at its own speed. It was however expected that the memory controller will operate at higher speeds than the arriving data so therefore the buffer was only expected to hold a couple of bytes of data as a result of addressing delays and high speed code word injection. The buffer was chosen to be larger than the input buffer at 32 bytes wide.

Decoder Input Buffer The main concern for this buffer was the arrival rate of bytes from the memory. This was expected to be controlled by the required download speed from the recipient device. In order to design, a buffer speed of 10M-bytes per second was chosen. This speed is near the maximum flash reading speed and therefore a fastest case scenario. In order to decode at such speed, it was chosen to implement a streaming decoder. This decoder inputs 3 entire code words before outputting the first symbols of the first code word. From then on code words are streamed in and streamed out at similar speeds. Therefore the input buffer can be fairly small. Simulation of this decoder showed that 2 symbols were the maximum depth required. By adding a safety margin, the total depth of the buffer was chosen to be 16 symbols.

Decoder Output Buffer Since the recipient of the data is unknown the amount of buffering will be trivial. The delays from the decoder will cause the outputted data speed to vary, therefore by buffering the recipient is given more control of the data arrival. Also, it would be desirable to output the entire codeword as fast as possible from the decoder once checked and corrected in order to increase the throughput rate to a maximum. This can be achieved in 1.28us. The buffer allows the data to be extracted from the decoder at this rate without stressing the timing of the recipient device. The output buffer must therefore be able to store an entire codeword. Adding a double safety margin, the decoder output buffer was chosen to be 128 bytes.

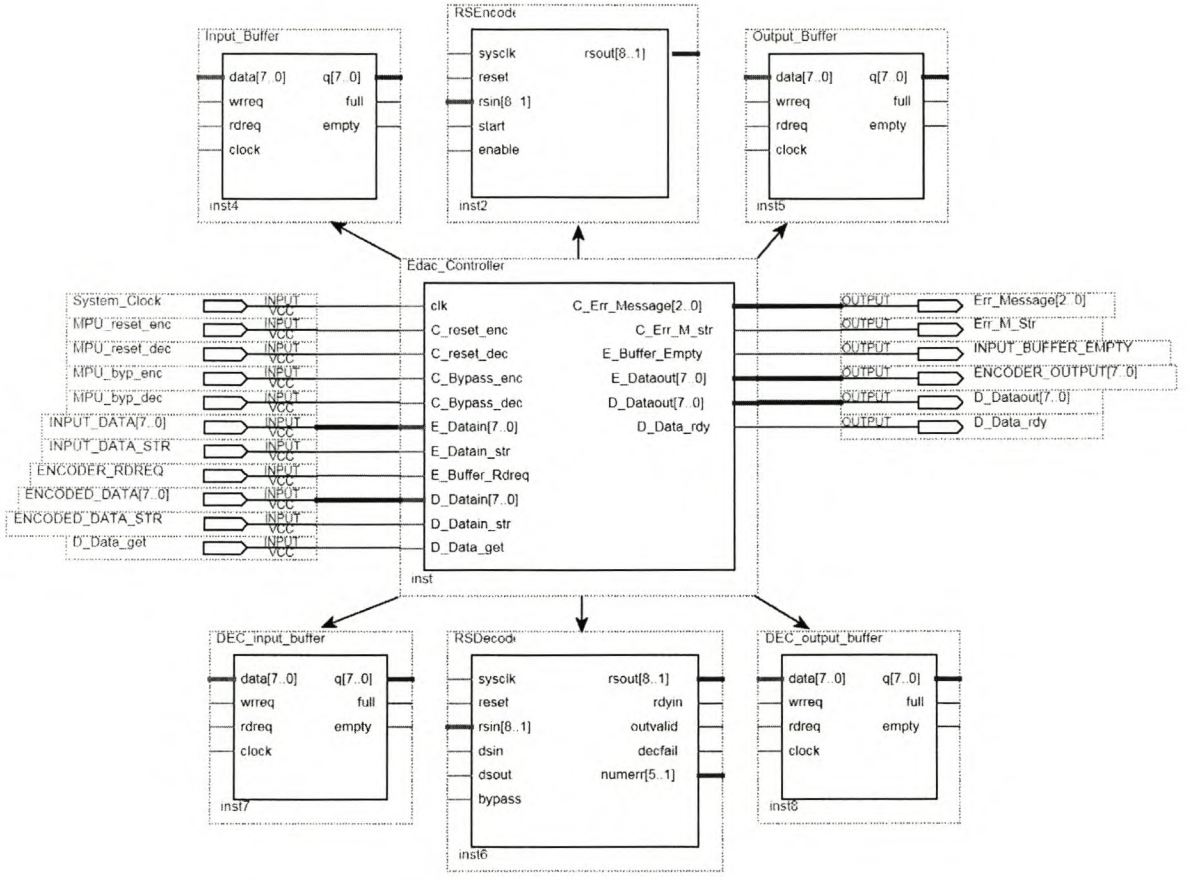


Figure 6.1: *EDAC system implementation in Quartus II*

EDAC Controller

The EDAC controller has to monitor and control the encoder, decoder, all the buffers and report to the master MPU controller. The concept circuit shown in figure 5.6 was realised in VHDL. Figure 6.1 shows the top-level entity, *Edac_Controller*, with all the embedded components within it. These components include the encoder, with its input and output buffers, and the decoder, also with its input and output buffers. The full set of VHDL code for this design can be found in appendix C.1.

Within the ***Edac_Controller*** block are the control processes which are the heart of the EDAC subsystem. These processes control all the signals which are connected to the EDAC_Controller and embedded components. Interaction with the master MPU controller is achieved through all the input and output *MPU_** lines. The MPU can instruct the EDAC circuit to reset or bypass the encoding or decoding systems by controlling the following lines: *MPU_reset_enc, MPU_reset_dec, MPU_byp_enc, MPU_byp_dec.*
Errors are reported to the MPU controller via the *MPU_error_message[2..0]* and *MPU_err_mes_str* lines. It was decided to give each of the possible errors a code which is then pulsed on these error lines to the MPU which will then react accordingly. The rest of the internal and external signals from the EDAC controller block are routed to either the encoder or decoder systems.

The ***RSEncoder*** block performs the Reed-Solomon encoding. Basically the encoder is controlled by initiating a new codeword using the *start* signal and then pulsing data symbols on the *rsin[8..1]* and *enable* lines respectively. The outputted symbols from the encoder are then sent through a multiplexer to the output buffer where they become available to the mass memory controller for storage. The multiplexer is used to bypass the encoder if required since the encoder does not have a bypass function as with the decoder.

Similar to the encoder, ***RSDecoder*** performs the Reed-Solomon decoding. As with the encoder, data is pulsed into the decoder via its *rsin[8..1]* and *dsin* input pins. Outputting data is controlled by the *dsout* signal and is outputted on the *rsout[8..1]* line. Unlike the encoder, the decoder has a bypass function which causes the inputted signals to be transmitted through the decoder without any adjustments. The *bypass* pin controls this operation. The rest of the decoder signals inform the controller about the status of the decoder such as : number of errors found, whether the decoder is ready to accept new signals, whether the outputted data is valid, and whether the current decoding process was successful.

The encoding and decoding processes were simulated in section 6.4.1 to confirm their

functionality.

6.3.2 SEL and SEFI design

The necessary concept designs for the SEFI and SEL system was developed in section 5.2.6 and section 5.2.7 respectively. These concepts include the power control circuit, the current monitoring circuit, the watchdog circuit, and the SEFI controller. Their respective low-level designs are implemented in the following subsections.

Power control

As mentioned in 5.2.6, the power control circuit must be able to supply and remove the 3.3 volt power rail required by the flash device. This circuit must also be controlled by the FPGA and implement in as little extra hardware as possible.

This circuit was realised by implementing a simple transistor switch. The characteristics required from the transistor were:

1. Low voltage drop across transistor since the supply voltage to the flash can range from 2.7V to 3.6V according to [14]. Therefore the voltage drop cannot be more than 0.6V.

2. Control via the FPGA. The current or voltage required to control the switching of the transistor must therefore be suppliable by the FPGA. According to [2], the input or output pins of the FPGA have the following absolute maximum ratings: Source or sink 25mA of current. Minimum and maximum output voltage of 3.0V and 3.6V respectively.

A small signal P-Channel MOSFET (metal oxide semiconductor field effect transistor) transistor was therefore chosen to meet these requirements. It was the most suitable choice since a MOSFET draws very little current from its control (or gate) pin and only has a small voltage drop across its source and drain pins. The circuit was implemented as shown in figure 6.2.
When the output pin of the FPGA is at logic level low (0V) the transistor is in an 'on' state and 3.3V is transferred to the flash device. If the FPGA output pin is brought to the high logic level (3.3V) then the transistor is switched off and the power rail supplied to the flash chip is brought down to 0V. Control over the power supplied to the flash is thus achieved.

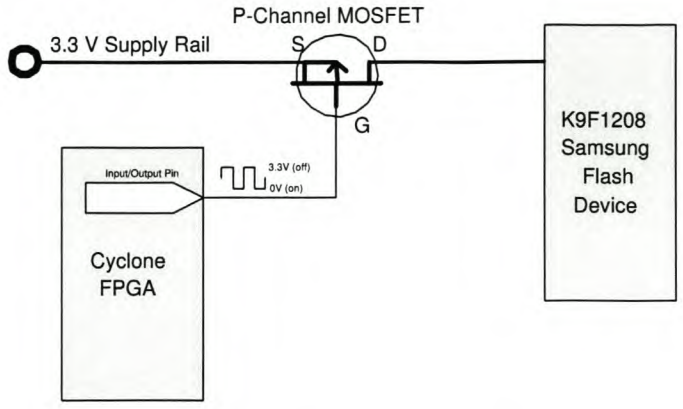


Figure 6.2: *Power control circuit*

Current Monitoring

As discussed in section 5.2.6, two external hardware devices are needed in order to monitor the current drawn by the flash device, namely an analog to digital converter and a current sensor. Firstly, a suitable current monitoring device must be chosen:

The MAXIM 4172 high side current-sense amplifier was chosen to monitor the current due to its availability, low cost and simple design. The chosen implementation circuit for the current sensor is shown in figure 6.4. As discussed in 5.2.6, the current is measured by amplifying the voltage across a small resister in series with the power rail. As shown in figure 6.4, two suitable resistor values must be designed for, namely R_{sense} andR_{out}. Their calculations are as follows:

- The differential input $V_{RS+} - V_{RS-}$, according to [9], has a maximum value of 0.3V and a typical value of 0.15V.
- The maximum current required to be measured,$I_{load}(max)$, is chosen as 100mA. This is not the highest expected current value due to latchup but a suitable value to distinguish when latchup is occurring. Therefore $R_{sense} = 0.15V/100mA = 1.5\Omega$
- The maximum input voltage to the A2D converter, according to [9], is calculated from the equation: $V_{out}(max) = V_+ - 1.2V$ therefore $3.3V - 1.2V = 2.1V = I_{out}(max)R_{out}$.
- The sensor equation is $I_{out} = 10mA/V \times (I_{load} \times R_{sense})$, therefore using maximum values becomes: $I_{out}(max) = 1 \times 10^{-3}R_{sense}$
- Therefore by substituting the previous two equations, $2.1V = 1 \times 10^{-3}R_{sense}R_{out}$. Which gives $R_{out} = 1.4K\Omega$.

The resistor values of 1.5Ω and 1.4KΩ were easily obtainable and therefore implemented.

The main factor in choosing an A2D converter is resolution. Since the analog signal has to be digitized into a binary number, certain inaccuracies occur due to the finite amount if bits. A higher number of bits gives a more accurate value. The resolution for an 8bit A2D converter is calculated as follows:

- 8 bits give 256 possible binary values.
- The voltage range to be measured is 0V to 3.3V therefore giving a resolution of 3.3V/256=12.9mV.
- If this value is converted back to the current through R_{sense},it gives a resolution of 614uA. which can be detected.

The main purpose of the current monitoring system is to distinguish when the flash device is off, on standby, operating and in failure. The difference in current drawn by these different states is in the order of milliamps. Therefore a resolution of 614uA is sufficient to distinguish between them. The AD7819 8bit A2D converter was therefore chosen.
It has a parallel interface, see figure 6.4, which is good in that the bit stream can be read without the need for serial to parallel conversion. The detailed operation of this device is documented in [3] and is controlled by the FPGA, see the VHDL entity in figure 6.3.

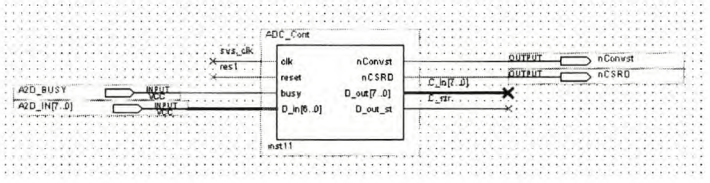


Figure 6.3: *Analog to digital converter controller*

Basically the A2D controller, in the FPGA, signals the A2D converter to begin a conversion. On completion, the controller is signaled by the converter that the process is complete and the digital value is then sampled by the controller via the connected bus lines. The value is then sent to the SEFI controller for any necessary processing. The full VHDL code for the A2D controller can be found in appendix C.5. The current monitoring system is shown in figure 6.4. The V_{ref} pin of the A2D converter shown in the figure is the voltage value that V_{in} is compared to when digitizing the analog signal. The ratio value of the latter to the former is equal to the digital output value compared to the maximum value of 255. In this case V_{ref} is equal to V_{cc} and therefore connected together.

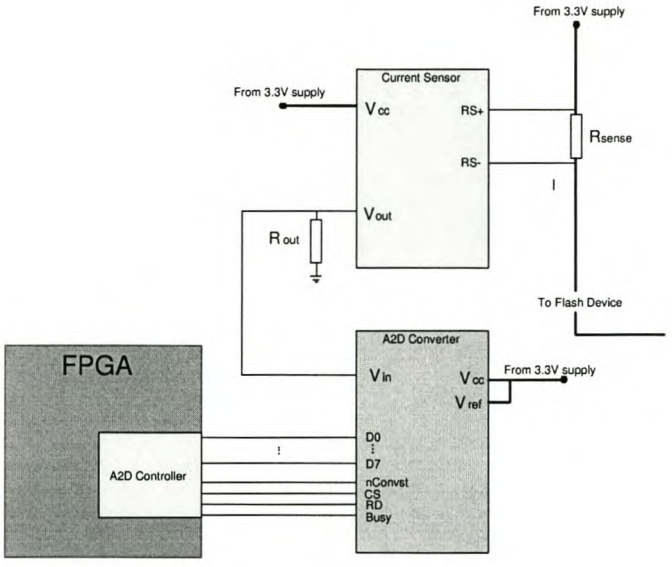


Figure 6.4: *Current monitoring system*

Watchdog Circuit

The watchdog circuit, as discussed in the concept design section, has to detect when the flash device has suspended. It was chosen to implement the watchdog timer in the FPGA. The watchdog entity developed in VHDL is shown in figure 6.5. The full VHDL code can be found in appendix C.4.

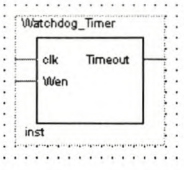


Figure 6.5: *Watchdog VHDL entity*

The watchdog is basically initiated by keeping the *Wen* signal high. If kept high for 1.34 seconds then the watchdog will set *timeout* high, signaling that the current process has timed out. The timeout period of 1.34 seconds is far longer than any single process time and will therefore be sufficient.

Control over the watchdog timer was chosen to be handled by a process within the SEFI controller which follows in the next section.

SEFI Controller

The SEFI controller is at the heart of the SEFI subsystem. In order to simplify the design, it was chosen to implement the SEFI controller and all the SEFI subsystem controllers as processes within one master MPU controller VHDL entity. This includes the watchdog circuit controller, power system controller, current analyzer, latchup monitoring, SEFI controller, and flash operation monitor. By implementing processes within one VHDL entity block, the design is simplified, in that extra signals and tri-state buffers linking the entities are no longer required. It also makes the VHDL design neater and easier to understand. The full VHDL for all the following processes can be found in appendix C.2.

Current Analyzer The incoming current values from the A2D controller is analyzed by the process **Current_Handler**. When the values are strobed in by the A2D controller, they are categorized into one of five levels: *cut_off, standby, operating, high, dangerous.* These current values can then be accessed by any of the processes which require current state knowledge.

Flash operation monitoring This is one of the most important processes since it monitors what state the flash device is in. As explained in subsection 5.2.6 this process monitors the physical control and bus lines connected to the flash device. As a new command is sent to the flash device, it must be captured and decoded by the flash operation monitor and then the other necessary processes informed, such as: setting the watchdog timer when an erase procedure begins, etc. This is achieved via the **Command_Handler** process. It categorizes the incoming command into one of seven main flash activities: *read1, readID, reset, pprog, cbprog, erase and status.* When categorized, the necessary states will inform the watchdog controller to monitor the current process.

Watchdog Controller The actual watchdog circuit was already implemented in a separate VHDL entity. The watchdog controller's duty is to operate the circuit accordingly. This is achieved via the **Watchdog_Handler** process. The Watchdog_Handler is controlled by the Command_Handler and also by the bus lines. When informed of a new flash process that requires monitoring, the watchdog initiates the watchdog circuit and then monitors that process until its completion. If it times out then the watchdog_handler will determine where in the current procedure the flash became unresponsive and inform the SEFI process accordingly.

SEFI controlling The SEFI process, **SEFI_Handler**, is responsible for determining what SEFI has occurred and controlling its mitigation. This process is initiated by a timeout from the watchdog circuit. It then determines which SEFI occurred by analyzing

what the current flash process was and what the current state is, and then mitigating the SEFI by either cycling power or resetting the flash device according to the mitigation solutions mentioned in 3.4.2. As mentioned in 5.2.6, resetting the flash device will be achieved by sending a reset signal to the mass memory controller, which should in turn send the reset command to the flash device.

Latchup monitoring The sudden rise of current to a dangerous level will indicate the occurrence of latchup. The Current_Handler process will detect this dangerous current level and, due to its destructive nature, inform the **Latchup_Handler** immediately. The latchup_handler is responsible for initiating a power cycle to the failing flash device and then monitoring whether it recovers from the latchup state. If permanently damaged then the power will be removed permanently from the flash and the master MPU controller will be informed.

Power system controller As shown in figure 6.2, the power circuit is controlled by an input / output pin from the FPGA. This was achieved by implementing the **POWER** process to control the selected pin. This process is responsible for cycling power to the flash device as well as permanently removing power to the damaged flash. This also includes ensuring that the switch is working correctly. Any power errors will also be reported to the master controller accordingly.

The above processes and their respective linking signals and busses are shown in figure 6.6. Note that the signals with a *Flash* prefix are from the physical bus lines and control lines, linking the mass memory controller and the flash device. The *F_power* signal from the *Power_handler* controls the MOSFET switching circuit and is therefore outputted from the FPGA via an input/output pin to the gate pin of the MOSFET. The *Send_reset_to_flash* signal outputted from the *SEFI_handler* is the necessary small amount of communication implemented between the mass memory controller and the MPU system. It instructs the mass memory controller to halt all operations and issue the *reset* command to the flash device.
Reporting to the master MPU controller is achieved by the *Error_message_to_MPU[2..0]* and *Error_power_switch* signals. These signals report which SEFI has occurred and if there is an error with the power circuit respectively.
This concludes the SEFI and SEL system implementation.

6.3.3 Bad Block design

The concept design for the bad block table was completed in section 5.2.8. Here the main goals for the system were stipulated and the main design sections were determined. In

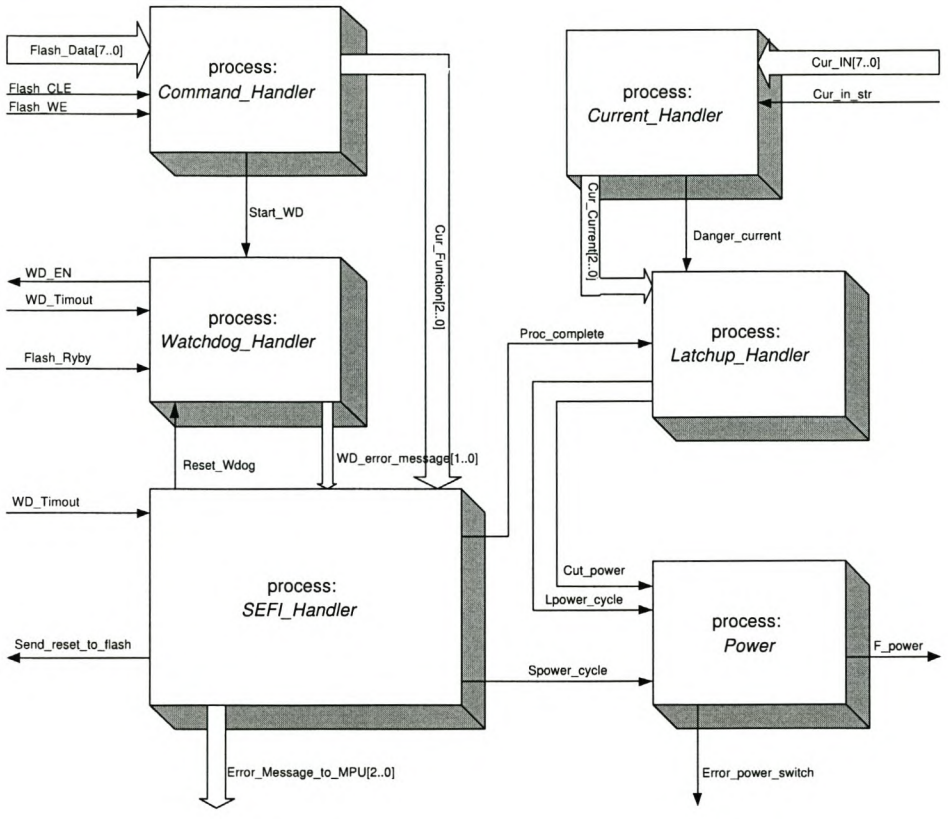


Figure 6.6: *SEFI processes and linking signals*

this section the main sections must be realised. These sections are : the RAMBLOCK, Address and Status decoder, and the Bad Block interface.

RAMBLOCK Design The conceptual layout of the proposed RAM space was shown in figure 5.8. In order to accommodate all the blocks in the K9F1208 Flash device, the table was increased to 4096 flag bit positions. As decided in 5.2.8 the table was implemented in the internal RAM of the Cyclone FPGA. The implemented RAMBLOCK is shown in figure 6.7 and has the following features :

- 4096 address blocks of 1 bit width. This gives each flag bit a unique address, therefore allowing instant access.
- Dual read/write ports for simultaneous updates and retrievals. This was necessary in case the mass memory controller was requesting block information while the table was being updated. By having one port for reading and one port for writing, access to the table will be guaranteed. There is the problem of simultaneous access to the same data, but this should not be a problem since the same block address will never be read while an update is occurring. This is ensured because updates only occur after a block has been accessed.
- Dual clocks with clock enable signals. This allows each port to be operated on without influencing each other and removes any clock skews.
- Registered input and output ports. This ensures synchronous input and output behavior.

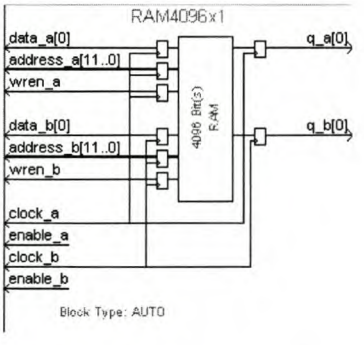


Figure 6.7: *The bad block table RAMBLOCK implemented in the FPGA*

To encapsulate the RAMBLOCK from the rest of the system and to meet its timing requirements, the RAMBLOCK was embedded as a component within a controlling VHDL entity, called ***BBCONTROLLER***. This VHDL block provides simple access to the RAMBLOCK and ensures that reading and writing to it is performed properly. The *BBCONTROLLER* VHDL entity is shown in figure 6.8.

Basically, data can be read by specifying the required address on *R_Adr[11..0]* and pulsing *R_Puls*. The corresponding data will then be pulsed out on *R_Dataout[0..0]* and *R_Strout*. Similarly data can be stored in the RAM by specifying the address, data and pulse on *W_Adr[11..0]*, *W_Datain* and *W_Puls* respectively. The full VHDL code for BBCONTROLLER can be found in appendix C.2.

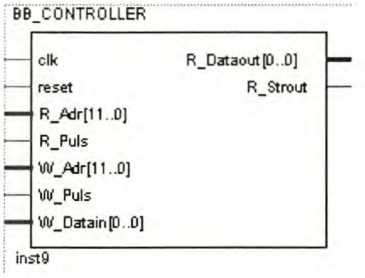


Figure 6.8: *The BBCONTROLLER VHDL block with embedded RAMBLOCK*

Address and Status decoder In order to update the bad block table when a new block becomes invalid, the status register must be read and the address of the current block must be known. The concept design of these decoders were discussed in subsection 5.2.8 and shown in figure 5.9. It entailed monitoring the physical bus lines connected to the flash.
It was therefore decided to implement these two decoders as VHDL processes within the master controller. This decision was made due to the fact that the SEFI processes (designed in the previous section), which utilize the same bus line resources, were implemented in this way. Therefore, by extending that design pattern, the overall design is simplified and all information obtained by the previous processes will be available to these processes. Furthermore, the BBCONTROLLER (with embedded RAMBLOCK) was also embedded within the MPU controller, therefore encapsulating the bad block table from view, which simplifies the design, and allowing easy access to the interface.

The duty of the address decoder is to monitor which block, within the flash memory space, is currently being used so that if it becomes invalid, the correct flag bit in the bad block table can be updated. This is achieved by monitoring the *Flash_ALE*, *Flash_WE* and *Flash_Data[7..0]* lines shown in figure 6.9. The complete address is then made available to the other processes. The decoder was realised via the **Adress_handler** process.

Similarly the status decoder reads in the status byte and checks the validity if the operation that was performed. The status decoder uses the information obtained from the Command_Handler process to determine when a status read has occurred. Although sta-

tus reads are not compulsory to operate the flash, their utilization is almost guaranteed since it is the most immediate way to monitor the operation. If an error is detected then the bad block interface is signalled to perform an update. The status decoder process is integrated into the bad block interface which is designed next.

Bad Block interface The duty of the bad block interface is to provide table access to the main memory controller and to perform the necessary updates. External 'read' access to the table was achieved by directly mapping the required read signals to the BBCONTROLLER component since no bad block reading is required by the MPU. However, 'write' access to the table is multiplexed between the external updates from the mass memory unit and internal updates from the MPU. External updates include loading the table with the stored table data in the flash, as explained in 5.2.8, and any other updates required by the mass memory unit. This was achieved by the **BB_Table_Handler** process. This process also incorporates the status byte checking mentioned in the previous paragraph. The actual initiating, loading and storing of the bad block table in the flash device is controlled by the mass memory controller, since no knowledge of the mass memory architecture is known. Reporting of invalid blocks is also realised by implementing a bad block counter process: **Bad_Block_Counter**. Its function is to count the total number of bad blocks and report it to the master MPU controller when appropriate.

The bad block processes, their linking signals and the embedded BBCONTROLLER component are shown in figure 6.9.

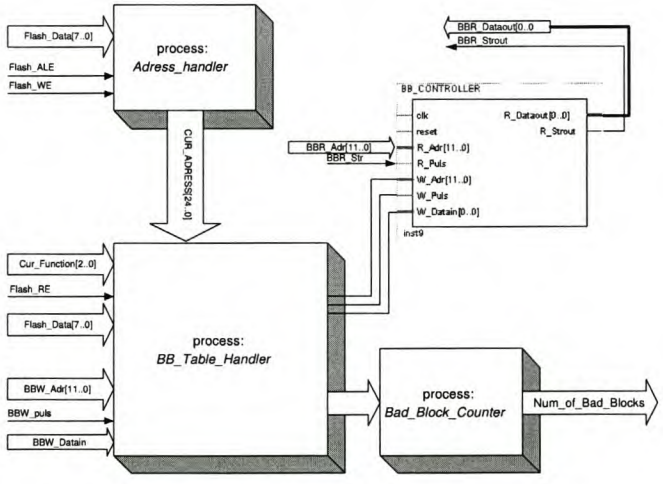


Figure 6.9: *The bad block processes and linking signals*

6.3.4 Master Controller design

As shown in figure 5.11, all the subsystems within the FPGA report to the master MPU controller. The duty of the controller therefore is to provide control to the systems that require it and to report malfunctions and errors to the OBC via the communications interface. Already the master controller VHDL entity consists of all the processes described in the previous sections, such as: all the SEFI and Bad Block processes. The MPU controller will therefore also be implemented as a process, or processes, within the entity, since most of the required linking signals are already available within it.

Reporting to the OBC It was decided to report any errors or failures to the OBC as soon as they occur. The process: ***MPU_error_handler*** obtains all error information from the subsystems and reports it to the OBC via the serial communication system, designed in the next subsection: subsection 6.3.5. Reporting to the OBC entails informing the OBC which type of error occurred and any extra information about the error. The error codes are shown in table 6.1.

Table 6.1: *Error reporting code protocol to OBC*

Error Type	**Error code**	**Extended Code**	
Bad block	0001		0000
SEFI	0010	read	0001
		program	0100
		erase	0110
Power Switch	0011		0000
EDAC	0100	Enc_bufin_full	0001
		Enc_bufout_full	0010
		Dec_decfail	0011
		Dec_bufin_full	0100
		Dec_bufout_full	0101
SEL	0101		0000
Power removed	0110		0000

The code is a byte in length and consists of the error-type-code as the most significant four bits and then either extra information or just zeros for the lowest significant four bits. After the code is sent, further information bytes are sent, such as the block address of the bad block, or the total accumulated bad blocks. The reporting is then ended by sending a stop byte, which consists of a byte of zeros.

Commands from OBC Commands can also be accepted from the OBC via the serial communication system. As with sending, the serial interface detailed in 6.3.5, informs the MPU that a command has been received from the OBC and the MPU must react accordingly. The process: ***OBC_Command_Handler*** controls the processing of new commands. Each command is represented by a code byte, shown in table 6.2.

Table 6.2: *Command Codes from OBC*

Command	Code
reset encoder	00000001
bypass encoder	00000010
reset decoder	00000100
bypass decoder	00001000
reset bad block table	00010000

These two processes complete the Master Controller and are shown in figure 6.10.

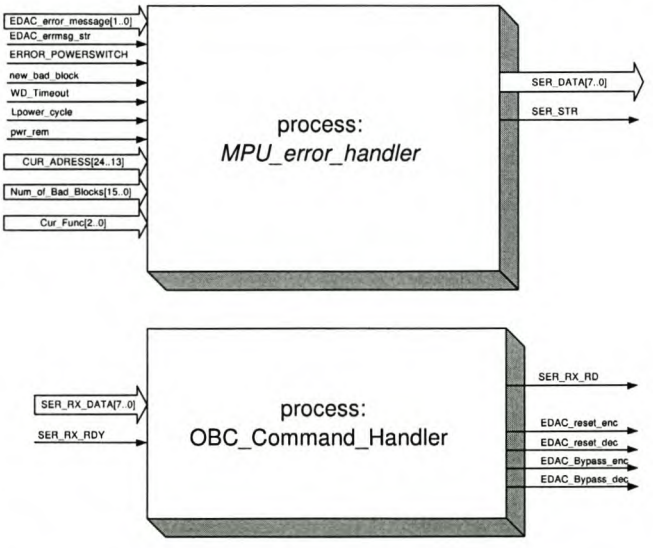


Figure 6.10: *The master controller processes and linking signals*

6.3.5 Error Reporting design

As discussed in section 5.2.9 a serial communication architecture is to be implemented. This entails creating an interface, designing a UART (Universal Asynchronous Receiver-Transmitter), and choosing and implementing an RS232 transceiver.

Interface and Uart design Due to the relatively slow speed of the serial communication compared to the 50MHz clock speed which the FPGA runs off, processes that require

to send bytes would have to wait in line for their turn. This would slow them down considerably which is unfavorable. To solve this problem a buffer will be implemented in which the bytes-to-be-sent can be queued in, which frees the process to continue, unaffected by the serial comm. Similarly when data is received from the OBC, it will be sent to an input FIFO buffer. The queued data can then be retrieved when suitable. The two buffer entities are shown in figure 6.11.

In order to test the serial communication, a personal computer (PC) will be used to simulate the OBC using a simple serial receive and transmit program. In order to send bytes to the PC it was deemed necessary to transform the intended binary byte of data into two bytes of hexadecimal-equivalent ASCII code (American Standard Code for Information Interchange). Therefore sending a binary byte of data results in two ASCII bytes being sent. This was due to the receiving program being effected by certain ASCII characters. The VHDL entity, ***bintohex_conv*** (shown in figure 6.11), was designed to complete this task.

The UART was also designed in VHDL. It is responsible for sending the outgoing data and receiving incoming data. The serial protocol decided upon was: Baud rate : 57600, Stop bits: 1, Data bits: 8, Parity: none. The bytes-to-be-sent are acquired from the binary-to-hexadecimal converter and then sent via the TX line to the PC. The UART also receives the input serial data stream via the RX line, converts it into a data byte and then sends it to the input buffer. The UART was realised by the VHDL entity: ***uart***, shown in figure 6.11. The BAUD rate clock controlling the serial transfer speed was created by the block : ***baudgen***. The full VHDL code for the component blocks in figure 6.11 can be found in appendix C.3

RS232 Transceiver Choosing a RS232 transceiver is not a major decision since the transceiver does not require any control logic and is not very expensive. It was decided to use the SP3232ECP transceiver from SIPEX due to its availability. The transceiver is responsible for converting the logic levels to RS-232 voltage levels and vice-versa.

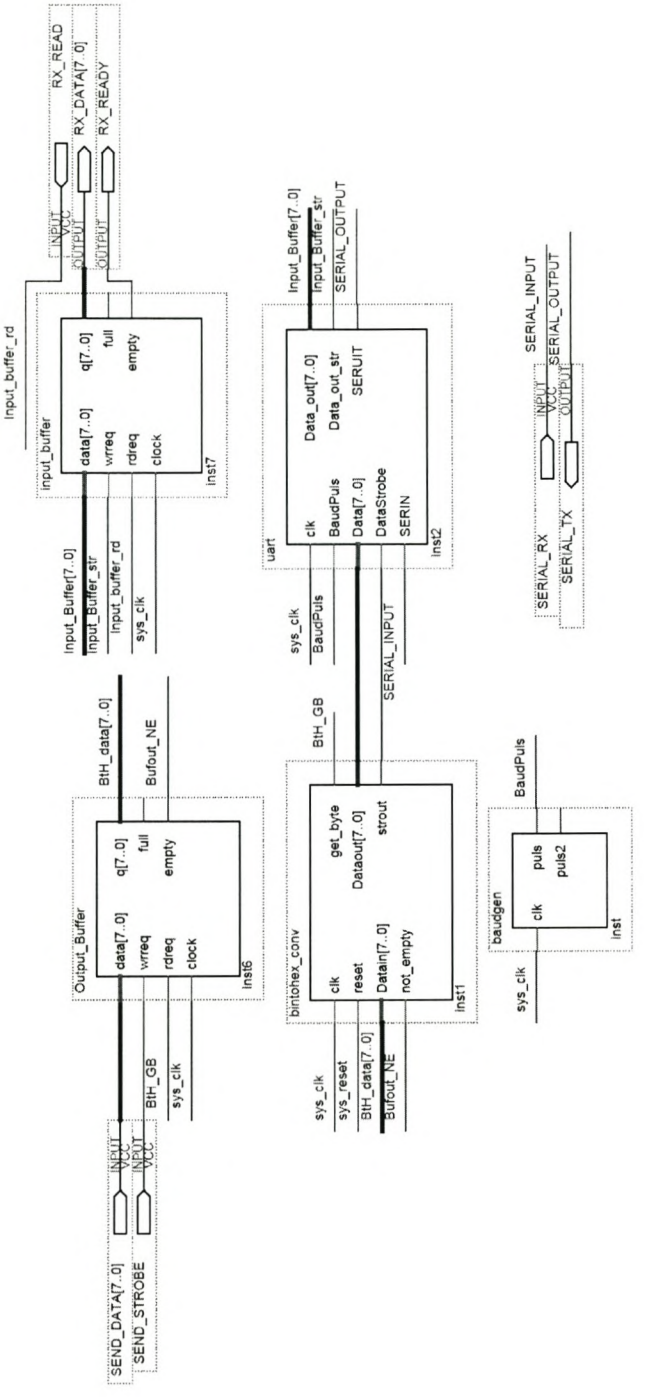


Figure 6.11: *Serial communication interface implemented in VHDL*

6.3.6 FPGA Choice and Power Calculations

Now that all the subsystems are complete, the entire system can be compiled together to calculate what size FPGA will be required and what power it consumes. The MPU VHDL entity is shown in figure 6.12. The input pins for all the various systems can clearly be seen.

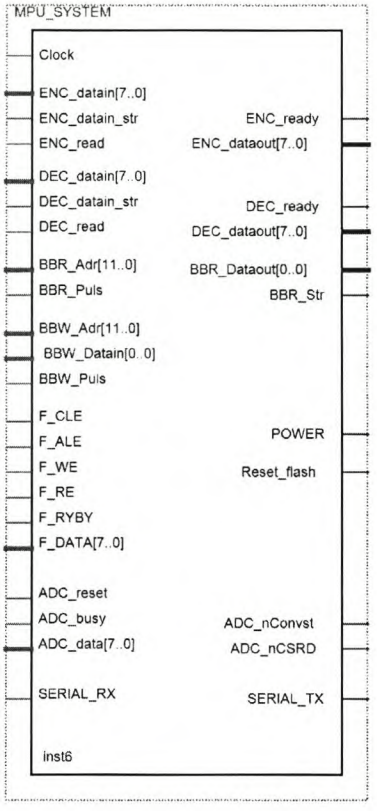


Figure 6.12: *Top layer entity for MPU system*

Compiling the entire system shows that the following resources are required: 4141 logic elements, 97 I/O pins, 16768 memory bits. The smallest ALTERA CYCLONE FPGA to fit this design is the EP1C6Q240C8 and would therefore be the most suitable choice. Compiling the design for this device gives a maximum clock frequency of 97.67 MHz which is more than sufficient to meet the 50MHz requirement.

Almost the entire system is implemented using the FPGA. Therefore most of the power required will be from the FPGA. Estimating the power required from it was achieved using the *Altera Cyclone Device Power Calculator Spreadsheet Version 2.0.* Basically,

the QUARTUS II software outputs a file which is analysed by the spreadsheet. A toggle percentage must also be specified, this represents how many pins will be toggling at one time. In order to calculate a worst case scenario, a toggle percentage of 50% was specified, which is reasonable considering all the pins which could be toggling. This gave a total power consumption of 712.36mW which is less than the specified limit of 1W. The power consumption by the FPGA is therefore acceptable.

The other hardware implemented was the A2D converter, the power switch, current sensor and the serial transceiver. The AD7819 analogue to digital converter typically consumes 10.5mW of power [3], therefore is not a major influence in the power consumption calculation. The MOSFET switch draws almost zero current and therefore not considered further. The current sensor draws a maximum of 1.6mA at 3.3V [9] giving a power figure of 5.28mW. The serial transceiver draws a maximum of 1mA [17] at 3.3V in this case, giving a power consumption of 3.3mW.
The combined power required is therefore 731.40mW which meets the 1W requirement set out.

6.4 Simulations

In order to verify that each of the designed subsections operate as required, each subsystem was simulated using the *Quartus II* simulator. This was achieved by emulating the operation of the flash device by controlling the return signals from it, such as its **RY/BY** (ready/busy) signal and data returned. Each of the various simulations are explained and documented to give the reader a fairly simple view of what is happening in the complicated signal environment.

6.4.1 EDAC

To verify the EDAC operation, firstly it was necessary to test the encoder with data arriving at 10 M-Bytes per second. Secondly the decoder was tested by using the encoded data from the previous test and inserting errors into that data stream. Lastly failures were instigated to test the error reporting capability of the EDAC system.

Encoding The encoding of a single codeword is shown in figure 6.13. A 50 MHz clock (20ns period) is inputted via the *System_Clock* signal. The four signals below it are control signals from the master controller. As designed for, data is inputted via the *INPUT_DATA* and *INPUT_DATA_STR* lines into the encoder input buffer at a rate of 10 M-Bytes/s. The start of the inputted data is shown at **A**. In order to make the inputted data easy to distinguish and to verify, it was chosen represent it as an increasing 8-bit

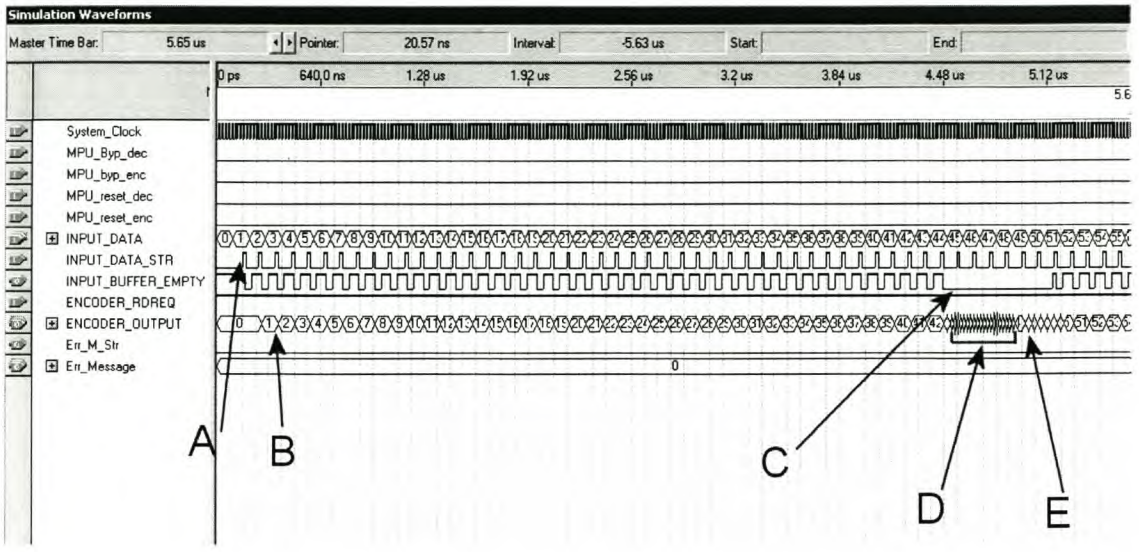


Figure 6.13: *Encoder Simulation*

counter value. The encoder output buffer's read request line, *ENCODER_RDREQ*, is set to 'high' therefore making the buffer transparent so its output represents the encoder's output, which is required. The time delay can be seen by the first inputted data byte at **A** and it being outputted from the encoder at **B**. **D** shows the code symbols being inputted into the data stream. As shown, the 21 code symbols are inputted after the last data byte of the codeword, byte 43. They are inputted at maximum speed to satisfy the timing constraints. While the codesymbols are being generated, the input buffer fills up (shown by *INPUT_BUFFER_EMPTY* signal at **C**) as it waits for the encoder to request the data bytes for the next codeword. When the next codeword is begun, shown at **E**, the queued data bytes are then sent to the encoder at maximum speed until the buffer is empty, then returns to the speed dictated by the arriving data. The last two signals show that no error has occurred.
This verifies that the encoding process meets the timing requirements and its functionality will be verified by the decoding simulation.

Decoding The decoding process is shown in figure 6.14. The control, error, and clock signals are the same as per the encoding simulation. Encoded data is created from the encoder and then fed to a noise generator which corrupts four of the codeword bytes, shown by the enlargement: **A**. This was done to verify the decoder functionality. The data stream, with errors, is then fed to the decoder input buffer at varying speeds (minimum speed of 10 M-Bytes per second). This tests the decoder's ability to handle data arriving at different speeds. As explained in the design section, 3 entire codewords must first be inputted into the decoder before the first bytes are outputted. As with the encoder

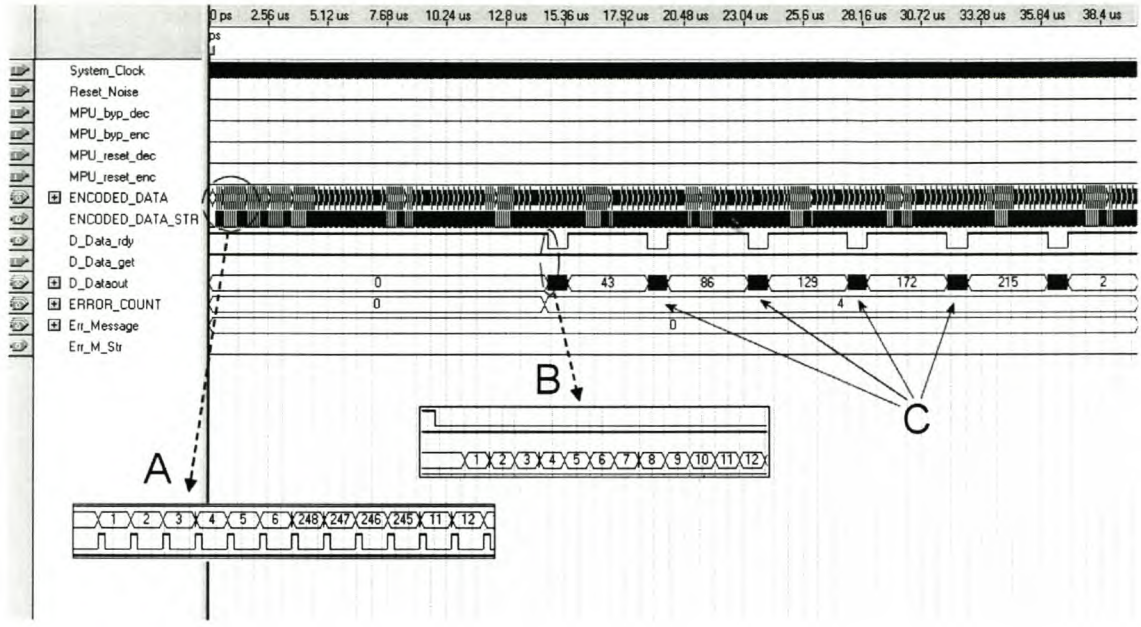


Figure 6.14: *Decoder Simulation*

simulation, the output buffer is made transparent by setting the read request signal to 'high'. The first decoded-bytes outputted on *D_dataout* and are shown by enlargement **B**. The four corrupted bytes have been corrected to their original values and the code symbols have been discarded. The number of corrected bytes are shown by the *ERROR_COUNT* signal, in this case 4 symbols are corrected in each codeword. Note that the corrected symbols from the decoder are outputted at maximum speed (every clock cycle) to meet timing requirements. After the entire first codeword's data-bytes have been outputted, the following codewords' data-bytes are outputted at regular intervals, shown by **C** (codeword 2,3,4 and 5). The *D_Data_rdy* shows when data is available at the output buffer. This simulation verifies the decoder functionality.

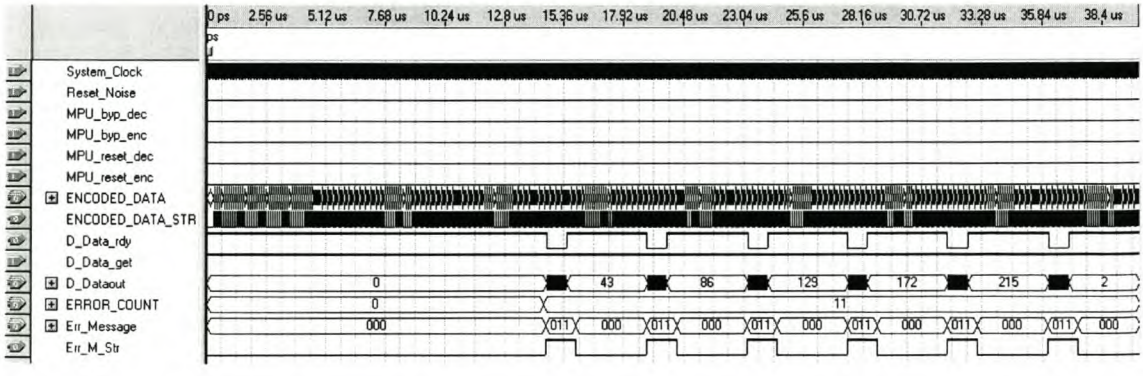


Figure 6.15: *Decoder failure Simulation*

To test the error failure reporting of the EDAC system, the number of corrupted symbols were increased to 11, which exceeds the correcting capabilities. The simulation is shown in figure 6.15. As with the previous simulation, the outputted symbols are placed on *D_dataout* except now the symbols are uncorrected since the maximum allowed errors were exceeded. The correct error code is placed on *Err_Message* and strobed on *Err_M_Str* to the master controller. All the other possible errors were tested and verified.

6.4.2 Bad Block

The bad block table was simulated by emulating the signals connected to the flash device. The simulation entails performing an erase operation to block 2497 then reading the status byte after completion. In order to view the entire simulation on one page, the block erase time was radically reduced. This does not influence the outcome since all the involved state machines merely wait for its completion. The simulation is shown in figure 6.16.

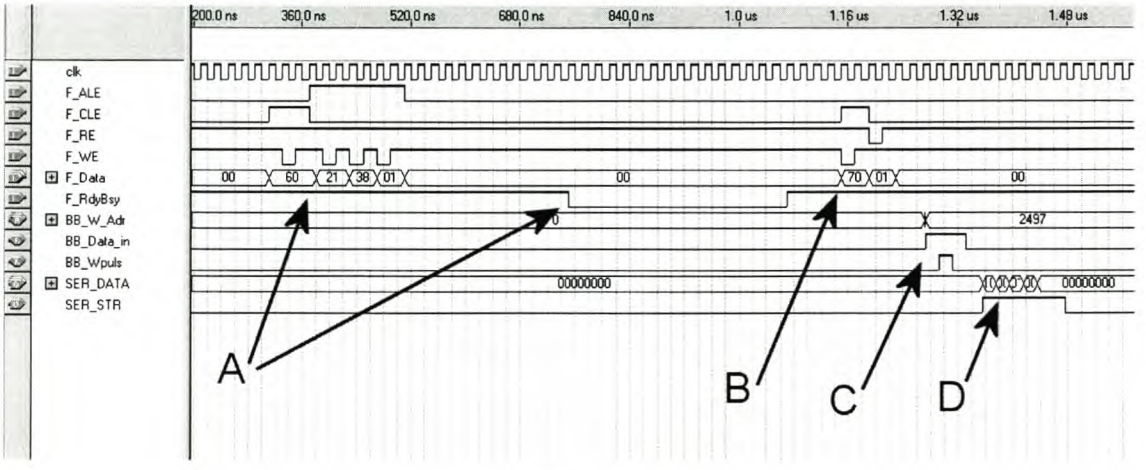


Figure 6.16: *Bad Block Table Simulation*

Label **A** shows the signals connected to the flash device. Firstly the erase code *60H* is written on the data line, then the block address *2497* in hexadecimal. The erase completion is signalled by the low-high transition of the *F_RdyBsy* line. The status read is then performed by writing *70H* to the flash device then reading in the data, *01H* in this instance, shown by **B**. This informs that the block erase was unsuccessful. The bad block table is then updated, shown by **C**. This involved setting the correct address and data on the *BB_W_Adr* and *BB_Data_in* lines respectively and then pulsing it through on the *BB_Wpuls* line. The bad block process is then completed by informing the OBC as mentioned in 6.3.4 via the serial port. The updated section of the table is shown in figure 6.17.

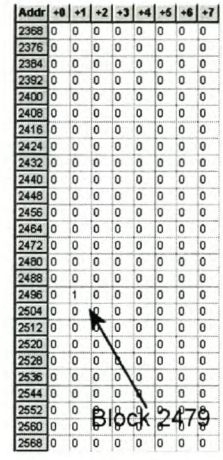


Figure 6.17: *Updated Bad Block Table*

6.4.3 SEFI

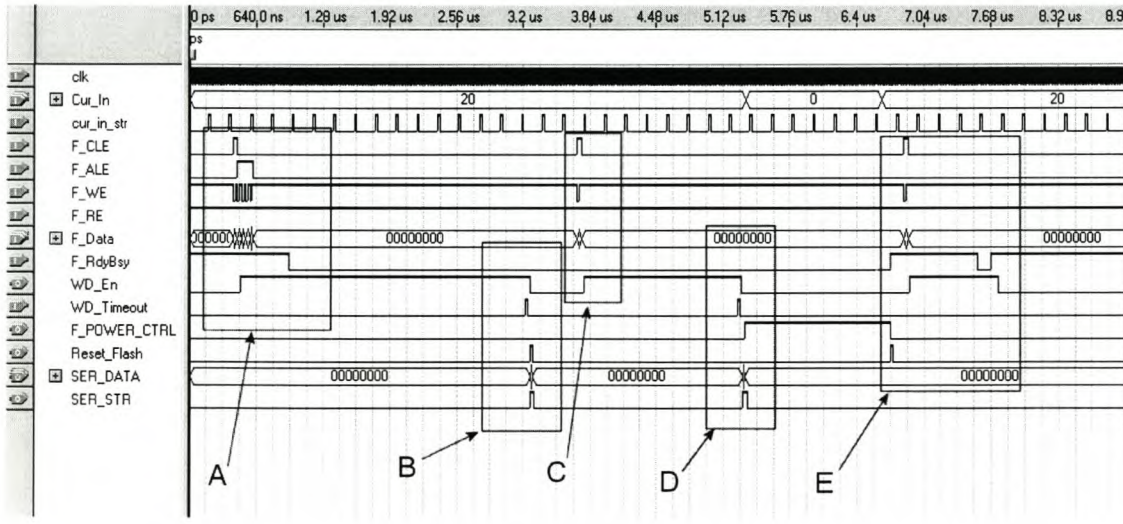


Figure 6.18: *Read SEFI Simulation*

To evaluate the SEFI handling system it was chosen to emulate a read operation that causes the flash not to respond. The simulation is shown in figure 6.18. As with the previous simulation, in order decrease the total time, the watchdog timer was decreased as well as the power cycle time.

Firstly, shown by **A**, the read command and address is sent to the flash which causes the watchdog enable (*WD_En*) to go 'high'. The flash ready-busy line (*F_RdyBsy*) is then emulated to go 'low' and get stuck in that state. Then in **B**, the watchdog timer times out (*WD_Timeout*) which causes the SEFI to be detected. The memory controller is then informed via the *Reset_Flash* signal and the OBC is also informed of the read-SEFI

via the serial port. The reset_flash signal causes the reset command to be sent, see **C**, which causes the watchdog to be set again. Since the ready-busy line is still emulated to be stuck low, the watchdog times out again, see **D**. Now, the OBC is informed of the reset-SEFI, shown in **D**, and then the power to the flash chip is cycled by setting the *F_POWER_CTRl* signal 'high' for a required period followed by another reset signal, see **E**. The power cycle causes the flash to return to an unstuck state therefore returning the *F_RdyBsy* to its 'high' state. The reset command is then sent to the flash chip which responds accordingly, showing its recovery from the read-SEFI.
This verifies the recovery operation caused by a 'read-suspend-SEFI'. The other types of SEFI handling were simulated in a similar way and verified.

6.4.4 SEL and Communication

It was decided to simulate the latchup handling and communication system together since latchup handling uses the communication system. The baud rate was greatly increased in order to make the simulation presentable, otherwise it would not have been possible to clearly show the control signals and the serial lines on the same simulation diagram.

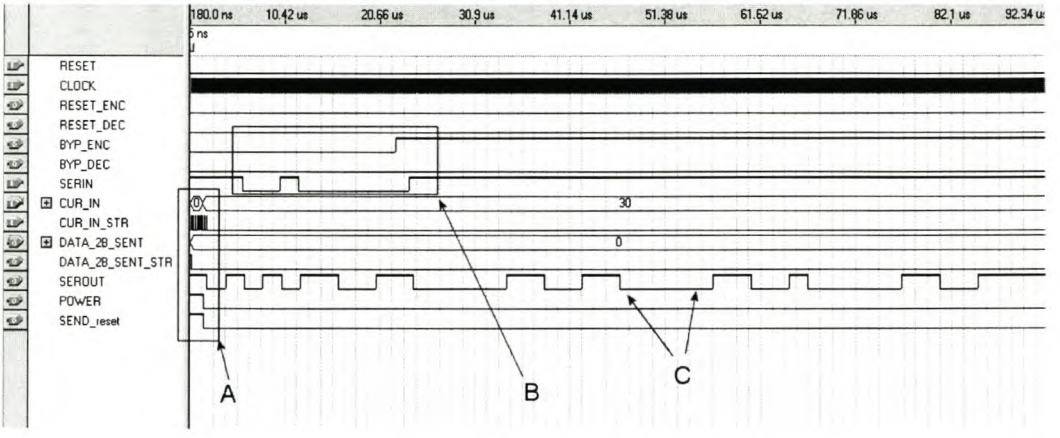


Figure 6.19: *SEL, Control and Communication simulation*

The simulation is shown in figure 6.19. Firstly, in **A**, latchup is easily simulated by inputting high current values from the A2D converter, this in turn results in the power being cycled and the 'send-reset' signal being pulsed. The OBC is then informed of the latchup via the correct commands being sent on the serial port. As expected, the four hexadecimal coded ASCII bytes were sent, shown by the *SEROUT* line and by marker **C**.

To test the serial input and command handler, the bypass-encoder command was sent to the MPU via the serial input line, *SERIN*, shown by **B**. The inputted data is then analysed and results in the *BYP_ENC* signal going high, as expected.

The other input commands from the OBC were simulated and their correct operation verified.

Chapter 7

Conclusion and Recommendations

In this chapter the achievements of this study are discussed and recommendations for further study and design are made.

7.1 What was achieved

A radiation test board and program was designed and assembled. This system was then radiated to investigate the effects of radiation on flash memory. The test yielded successful results which gave valuable insight into the problems that had to be overcome. To date, the test board and program is still being used to perform radiation tests on different types of flash memory.

Using the results from the radiation test and by defining rough requirements, a set of engineering specifications were produced. These specifications went through a thorough design process which resulted in the memory protection unit (MPU) being realised. The operation of the various subsystems in the MPU were verified by successful simulations. This showed that the various possible errors caused by radiation, such as bit flips, stuck bits, latchup, functional errors and bad blocks, could be detected, maintained, mitigated and reported. It also verified that 1 bit per byte of data could be protected and that it could be achieved at a data rate of 10Mbytes per second.

The design methodology allows the MPU to be added to any flash MMU, with minimal reconstruction. This is because it was developed to be almost transparent to the original memory system, except for a few linking signals. The power required in implementing this system was within the restrictions and therefore showed that it is a feasible option to use onboard a satellite.

The project went as far as to design the system to monitor one flash device. However, a

modular approach was maintained throughout the study and therefore, certain sections can easily be duplicated to monitor more than one device.

Different Implementation Settings In order to alter the MPU to protect a different size flash device, would entail making small changes to the system. This includes altering the following signals:

- *BBadrWidth* - Bit width of total number of blocks in flash device.
- *AdrWidth* - Bit width of total adress to flash device.
- *PageWidth* - Bit width of page number per block.

And will also require reconfiguring the size of the bad block table within the FPGA memory. This can be achieved by editing the existing table using the *Megafunction Plug-In Manager* found in *Quartus II*.

7.2 Recommendations

1. This design was evaluated by simulating the proposed systems while emulating the flash device's behavior, in software. In order to achieve a more accurate and solid result, it is suggested to implement the design in a PCB and test it, using faulty flash devices or by emulating a faulty flash device.

2. This study focused on protecting a single flash device, however, most mass memory systems consist of arrays of memory devices. Therefore it is suggested to duplicate and expand the MPU to incorporate the entire mass memory system. Certain sections of the MPU would not require duplication, such as the EDAC system, which would keep the MPU system small enough to be feasible.

3. A certain amount of interaction between the MPU and mass memory controller was inevitable. It is suggested to design a mass flash memory storage unit which incorporates the MPU to achieve a protected MMU.

4. To give the MPU control over the flash device, the bus lines between the MMU and flash device could be passed through the MPU. This would allow the MPU to manipulate the signals sent to, and received from the flash. Therefore, the activities of the flash could be monitored as before, as well as manipulated if required. This would give the MPU a wider range of protection options and would decrease its dependency on the mass memory controller.

Bibliography

[1] ALTERA. *ReedSolomon Megafunction for Altera FPGA*, 2002.

[2] ALTERA. *Cyclone FPGA Family Data Sheet*, 2003.

[3] ANALOG DEVICES. *+2.7V to +5.5V,200kSPS 8-Bit Sampling ADC*, 2000.

[4] BADENHORST, P. J., "Module for the second generation SUNSAT micro-satellite." Master's thesis, University of Stellenbosch, 1996.

[5] CLARK, G., *Error-correction coding for digital communications*. New York : Plenum Press, 1981.

[6] GROBLER, H., "Aspects Effecting the Design of a Low Earth Orbit Satellite On-Board Computer." Master's thesis, University of Stellenbosch, 2000.

[7] HASS, K. J. *et al.*, "Mitigating Single Event Upsets From Combinational Logic." *7th NASA Sympoaium on VLSI Design*, 1998, pp. 4.1.1–4.1.10.

[8] HOLMES-SIEDLE, A. and ADAMS, L., *Handbook of radiation effects*. Oxford: Oxford University Press, 2002.

[9] MAXIM. *Low-Cost, Precision, High-Side Current-Sense Amplifier*, 2002.

[10] MESSENGER, G. C. and ASH, M. L., *The Effects of Radiation on Electronics, second ed.*. Van Nostrand Reinhold, New York, 1992.

[11] NGUYEN, D., GUERTIN, S., SWIFT, G., and JOHNSON, A., "Radiation Effects on Advanced Flash Memories." *IEEE Transactions on Nuclear Science*, December 1999, Vol. 46, No. 6, p. 1744.

[12] NGUYEN, D. and SCHEIK, L., "SEE and TID of emerging non-volatile memories." Work carried out by Jet Propulsion Laboratory, California Intitute of Technology, 2001.

[13] PHILIPS SEMICONDUCTORS. *The I^2C-bus and how to use it*, 1995.

[14] SAMSUNG. *K9F1208 64M × 8 Bit, 32M × 16 Bit NAND Flash Memory*, 2003.

[15] SCHWARTZ, H., NICHOLS, D., and JOHNSTON, A., "Single-Event Upset in Flash Memories." *IEEE Transactions on Nuclear Science*, December 1997, Vol. 44, No. 6, p. 2315.

[16] SHIRVANI, P., NIRMAL, S., and MCCLUSKEY, E., "Software-implemented edac protection against seus." Stanford University, 1999.

[17] SIPEX. *True +3.0V to +5.5V RS-232 Transceivers*, 2001.

[18] TOSHIBA. *NAND Flash Aplications Design Guide*, 2003.

[19] TYSON, J., "How Flash Memory Works." `www.computer.howstuffworks.com/flash-memory.htm/printable`. July 2003.

[20] UNIVERSITY, R., "Error Correction with Hamming Codes." `www2.rad.com/networks/1994/err_con/hamming.htm`. 1994.

Appendix A

Radiation Test Board

A.1 Design Schematic

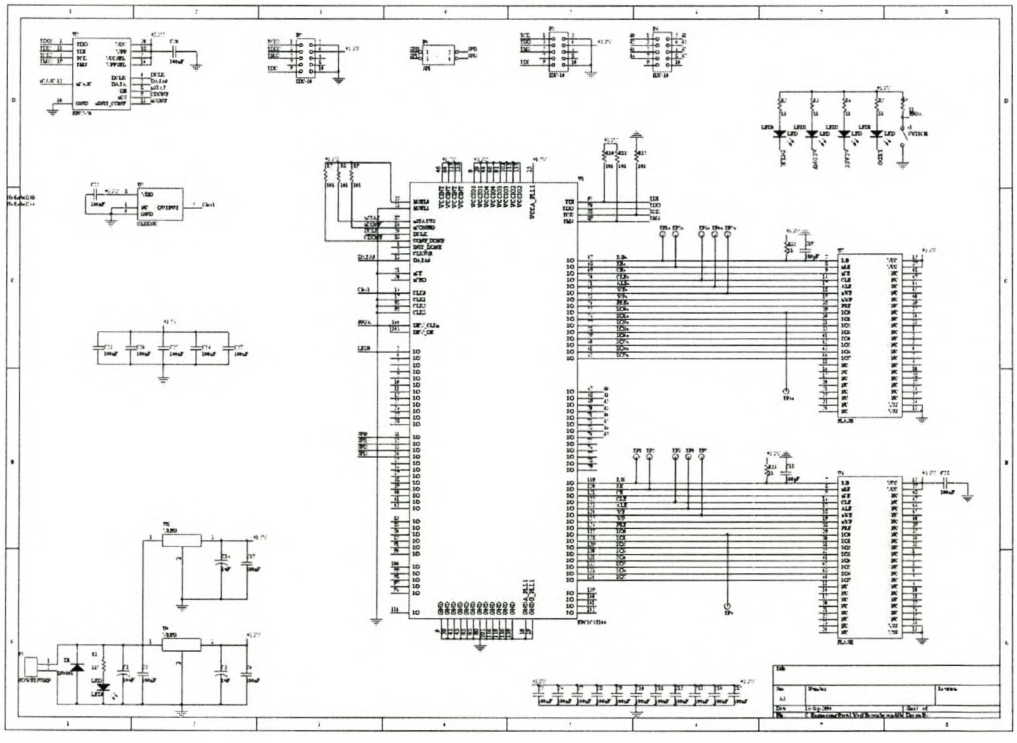


Figure A.1: *Radiation Test Board Schematic*

A.2 Printed Circuit Board (PCB) Layout

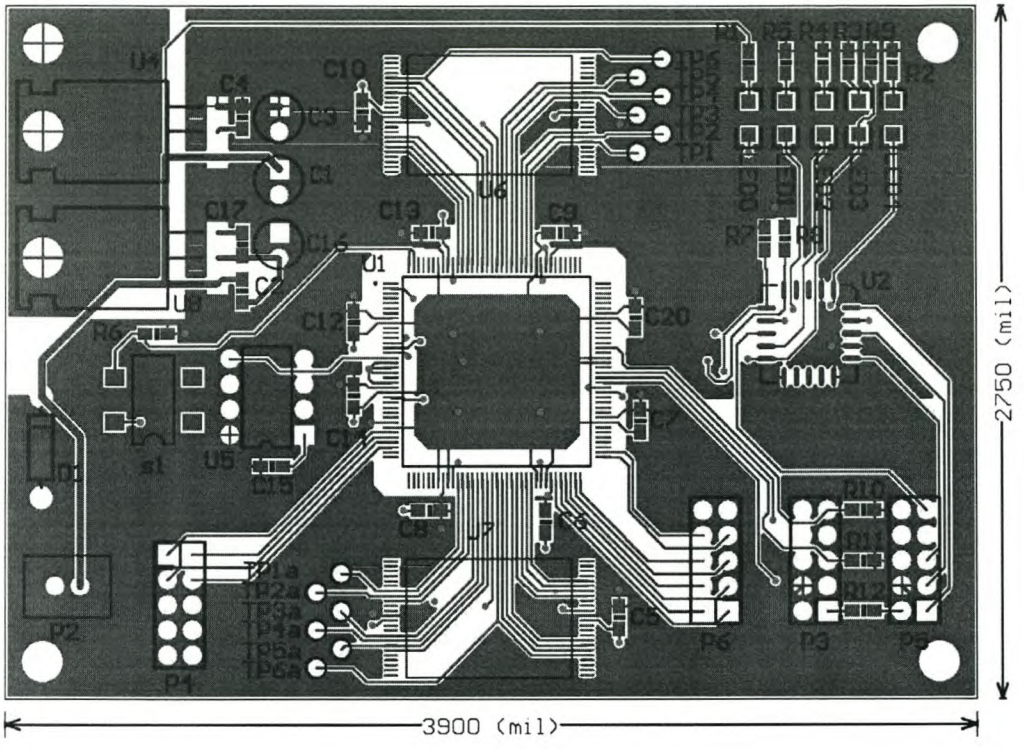


Figure A.2: *PCB Front Layout*

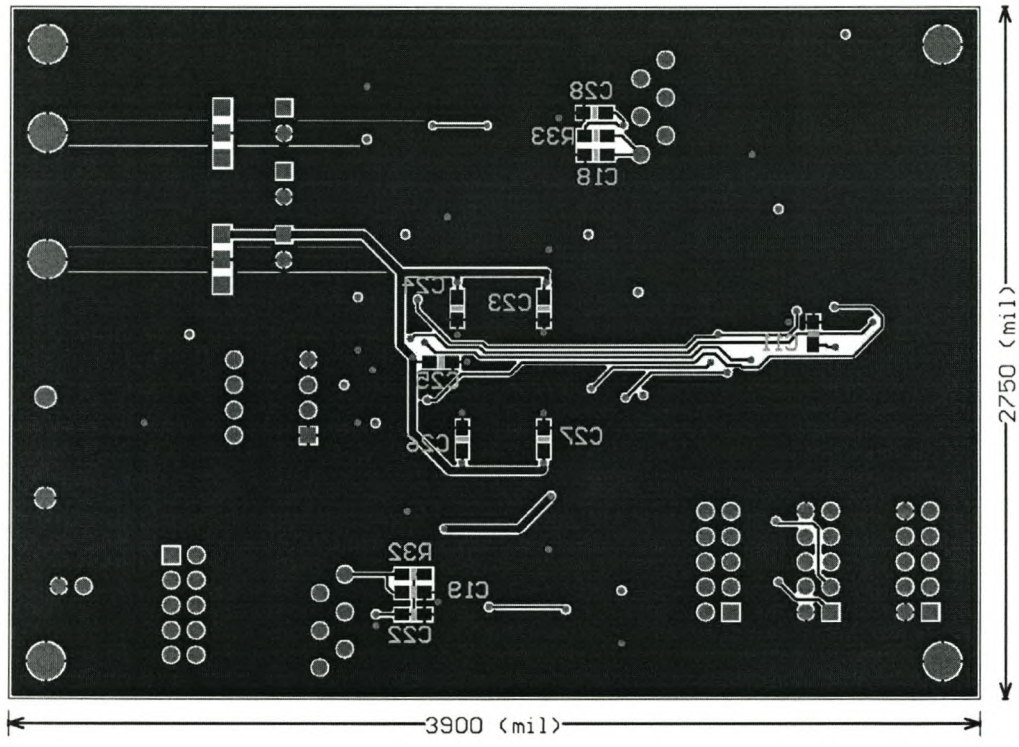


Figure A.3: *PCB Bottom Layout*

Appendix B

MPU Signal Listings

The signals used in the MPU system are listed here. This includes the signals name, width and description. Refer to these descriptions when studying the low level designs in 6.3, the simulations in 6.4 or the VHDL code in appendix C.

B.1 EDAC Controller

B.1.1 Input/Output Signals

Table B.1: *General and MPU Report and Control I/O*

Signal	Width/Type	Description
clk	bit	This is the system clock that drives all the state machines. It is designed for a 50MHz input clock.
C_reset_enc	bit	This signal from the master controller informs the EDAC controller to reset the encoder.
C_reset_dec	bit	" Informs the edac controller to reset the decoder.
C_bypass_enc	bit	" Informs the edac controller to bypass the encoder.
C_bypass_enc	bit	" Informs the edac controller to bypass the decoder.
C_Err_Message	3 bits	The error bus informs the master controller what error has occurred.
C_Err_M_str	bit	Informs the master controller that an error has occurred.

Table B.2: *Encoder I/O*

Signal	Width/Type	Description
E_Datain	8 bits	Incoming data bus for data to be encoded and then stored in memory
E_Datain_str	bit	Indicated when new byte has arrived
E_Buffer_Rdreq	bit	Signal for memory controller to request byte from encoder output buffer
E_Buffer_Empty	bit	Informs memory controller when data is available in output buffer
E_Dataout	8 bits	Output data bus from encoder output buffer to memory controller

Table B.3: *Decoder I/O*

Signal	Width/Type	Description
D_Datain	8 bits	Input data bus linking the memory controller and the decoder input buffer.
D_Datain_str	bit	Strobe signal indicating new byte has arrived at decoder input buffer
D_Dataout	8 bits	Output data bus linking the decoder and the peripheral system which requested the data. Decoded data bytes are sent via this bus
D_Data_rdy	bit	Ready signal informing peripheral system that decoded data is available in output buffer
D_Data_get	bit	Request signal for peripheral system to obtain decoded data from the buffer

B.1.2 Internal Signals

Table B.4: *Signals linked with Encoder stage*

Signal	Width/Type	Description
symbol_num	integer	counts the number of symbols within the codeword
Bufin_full	bit	interfaces with input buffer, informing when buffer is full
Bufin_empty	bit	interfaces with input buffer, informing when bytes are available
Bufin_rdreq	bit	interfaces with input buffer, controls reading of data from the buffer
Bufin_dataout	8 bits	data bus between input buffer and encoder
Enc_reset	bit	links reset input pin to encoder
Enc_enable	bit	controls inputting of data into the encoder
Enc_start	bit	signals the beginning of a codeword
Enc_Dataout	8 bit	data bus between encoder and multiplexer
Bufout_full	bit	interfaces with output buffer, informing when buffer is full
Bufout_Wrreq	bit	controls writing of data into output buffer
Muxout	8 bit	outputs the selected data bus to the output buffer
Muxout_str	bit	outputs the selected data bus strobe to output buffer

Table B.5: *Signals linked with Decoder stage*

Signal	Width/Type	Description
Dec_reset	bit	resets decoder
Dec_out	bit	signal controls the outputting of symbols from decoder
Dec_rdyin	bit	informs when decoder is ready to accept new symbols
Dec_outvalid	bit	signals when data is available from the decoder
Dec_decfail	bit	informs whether the decoding process was successful or not
Dec_numerr	5 bits	this data bus shows how many symbols were in error with the last decoded codeword
Dec_rsout	8 bits	data bus linking the decoder and the decoder output buffer
Dec_dsin	bit	controls the inputting of data into the decoder
Dbufin_rdreq	bit	controls the reading of data out of the decoder input buffer
Dbufin_empty	bit	signals when data is available from the decoder input buffer
Dbufin_full	bit	signals when the decoder input buffer is full
Dbufin_q	8 bits	data bus linking the decoder and its input buffer
buffer_strobe	bit	controls data flow between decoder and output buffer. Discards code symbols from data stream
Dbufout_full	bit	signals when output buffer is full
rcnt	integer	counts outputted symbols in each codeword to inform when code symbols can be discarded

Table B.6: *Encoder Control State Types*

state_type0	Description
reset0	Reset state, all signals are set to their reset values. A reset event or end of a codeword forces this state.
start0	Start state, this state signals the beginning of a codeword to the encoder
s1	This state waits for an available byte at the input buffer then sends it to the encoder. It also checks for the last required data byte for the codeword.
s2	Here the new byte is pulsed into the encoder
s3	The code symbols are outputted from the encoder in this state. The end of the codeword is also signalled by this state

Table B.7: *Decoder Control State Types*

state_type1	Description
reset1	Here all signals are set to their reset values. A reset event forces this state.
wait_rdy	This state waits for data to arrive at the decoder and also for the decoder to be ready. When both are satisfied, data is requested from the decoder input buffer
start1	Here data is inputted into the decoder

B.2 Master Controller

B.2.1 Input/Output Signals

Table B.8: *General and Misc I/O*

Signal	Width/Type	Description
clk	bit	50 MHz system clock input pin
ResetBB	bit	signal to reset state machines in BB_Controller component and reset bad block counter
Cur_In	8 bit	This data bus transfers the current value between the ADC and the MPU
cur_in_str	bit	Informs when a new current value is set
F_POWER_CTRL	bit	This signal controls the power circuit, either switching the power to the flash off or on
Reset_Flash	bit	The reset signal connects to the memory controller, informing it to send the reset command to the flash
WD_En	bit	Watchdog enable signal which controls the watchdog circuit operation
WD_Timeout	bit	Watchdog timeout signal informs the MPU when the watchdog has timed out

Table B.9: *Flash device Inputs*

Signal	Width/Type	Description
F_CLE	bit	command latch enable, signals when a new command is set
F_Data	8 bits	data bus to and from flash device
F_RdyBsy	bit	ready-busy, signals when flash is ready or busy
F_ALE	bit	address latch enable, signals when an address is being set
F_WE	bit	write enable, clocks data to the flash
F_RE	bit	read enable, clocks data out of the flash

Table B.10: *Bad Block I/O*

Signal	Width/Type	Description
BBR_Adr	11 bits	read address, sets the required address to be read
BBR_Puls	bit	signals that address has been set
BBR_Dataout	bit	read data bus from required address
BBR_Strout	bit	signals requested data available
BBW_Adr	11 bits	write address, set required address to be written to
BBW_Puls	bit	signals that required address and data has been set
BBW_Datain	bit	write data bus

Table B.11: *Serial Communication I/O*

Signal	Width/Type	Description
SER_DATA	8 bit	serial output data bus
SER_STR	bit	signal to strobe data into output-send buffer
SER_RX_DATA	8 bit	serial input data bus
SER_RX_RD	bit	read request signal to serial input buffer
SER_RX_RDY	bit	signal informing serial data has arrived

B.2.2 Internal Signals

Table B.12: *Watchdog controller process signals*

Signal	Width/Type	Description
F_ryby	bit	synchronized ready/busy signal
proc_complete	bit	signal informing when a flash device process was successful
WD_Error_Message	2 bits	error bus to SEFI process informing why timeout occurred

Table B.13: *Command and Current process signals*

Signal	Width/Type	Description
CUR_FUNC	3 bits	current function register which is set to a 3 bit code representing the current, or last executed, flash operation
Start_WD	bit	this signal informs the watchdog to begin counting when a new command arrives at the flash device that requires monitoring
CUR_CURRENT	3 bits	current current register which is set to a 3 bit code representing the last current state inputted by the ADC. The different states can be seen by the constant declarations
danger_cur	integer	counts the number of consecutive high current values. It is used to distinguish current spikes from latchup currents

Table B.14: *SEFI process signals*

Signal	Width/Type	Description
send_reset_to_flash	bit	informs memory controller to send reset command
reset_wdog	bit	resets watchdog process to idle state
Spower_cycle	bit	instructs power circuit to cycle power to flash device

Table B.15: *Latchup and Power process signals*

Signal	Width/Type	Description
cut_power	bit	signals power circuit to permanently remove power from flash device
pwr_rem	bit	informs error handler that power has been removed from flash device
Lpower_cycle	bit	instructs power circuit to cycle power to flash device
ERROR_POWERSWITCH	bit	informs error handler that an error has occurred with the power switch
F_POWER	bit	holds current state of power switch

Table B.16: *Address handler and bad block handler signals*

Signal	Width/Type	Description
CUR_ADRESS	25 bits	this data register holds the current, or last used, flash device memory address
adress	25 bits	this register holds the flash address while being read in
new_bad_block	bit	reports to the error handler when a bad block has been detected
Num_of_Bad_Blocks	16 bits	holds the total number of bad blocks
temp_status	bit	temporarily holds the status byte before being checked

B.3 Communication

B.3.1 Input/Output Signals

Table B.17: *Baudgen I/O*

Signal	Width/Type	Description
clk	bit	50 MHz system clock
puls	bit	BAUD rate output signal. This signals controls the UART
puls2	bit	a 16× slower baud rate signal

Table B.18: *Bintohex_conv I/O*

Signal	Width/Type	Description
clk	bit	50Mhz system clock
reset	bit	general reset signal to reset state machine
Datain	8 bits	input data bus linking output data buffer and Bintohex_conv
not_empty	bit	signal informing when data is available to be sent
get_byte	bit	requests data from output buffer
Dataout	8 bits	output data bus to UART
strout	bit	signals UART that new data available

Table B.19: *UART I/O*

Signal	Width/Type	Description
clk	bit	50Mhz system clock
BaudPuls	bit	BAUD pulse from Baudgen
Data	8 bits	input data bus from Bintohex_conv
DataStrobe	bit	strobe line from Bintohex_conv
SERIN	bit	Serial input line
Data_out	8 bits	input data bus to input data buffer
Data_out_str	bit	input data strobe line to buffer
SERUIT	bit	Serial output line

B.3.2 Internal Signals

Table B.20: *Baudgen State Types*

state_type	Description
counting	this state clears the strobe signals and counts the clock signals until the baud rate is reached
high	this state either outputs a strobe on *puls* or signals a double pulse
double	here a strobe is placed on both the *puls* and *puls2* output signals

Table B.21: *Bintohex_conv State Types*

state_type	Description
idle	this state waits for a byte to become available from the output buffer then requests it
getbyte	clears request line
getbyte2	this state splits the requested byte up and converts it to an integer
send1	here the two integers are converted to binary equivalent ASCII code
send2	in this state the first of the two bytes is sent to the UART
send3	this state waits for the UART to send the first byte
send4	the second byte is set here on the output bus
send5	this state sends the second byte to the UART
send6	this is a wait state for the second byte to be sent

Table B.22: *UART State Types*

state_type_T	Description
idle	this state wait for a new byte to be strobed to the UART for sending.
xmit	this state transmits the serial data packet on the serial output line at the supplied baud rate.
state_type_R	**Description**
idle	here the data received signals are cleared and the serial receive line is also monitored for arriving data.
rec	this state reads in the serial data packet and paralyzes the data bits.
send	here the received data byte is sent to the input buffer.

B.4 Watchdog Timer

B.4.1 Input/Output Signals

Table B.23: *Watchdog_timer I/O*

Signal	Width/Type	Description
clk	bit	50MHz system clock input
Wen	bit	watchdog enable signal that controls operation of watchdog timer. The timer counts when this signal is high. When it is low, the timer is cleared and stays idle
Timeout	bit	the timeout signal pulses when the watchdog times out

B.4.2 Internal Signals

Table B.24: *Watchdog State Types*

state_type	Description
counting	this state is forced when *Wen* is goes high. It counts the positive clock edges of the input clock.
timeup	this state sets the timeout signal
wait_reset	here the timeout signal is cleared and the state remains here until reset

B.5 Analog to digital converter

B.5.1 Input/Output Signals

Table B.25: *ADC_Cont I/O*

Signal	Width/Type	Description
clk	bit	50MHz system clock input
reset	bit	global reset signal, resets state machine
nConvst	bit	Conversion Start. by pulsing this signal, the ADC begins a conversion process
nCSRD	bit	Chip Select/ Read. this signal controls reading out the digital data
busy	bit	this signals shows when the ADC is busy in the conversion process
D_in	8 bit	data in bus. this bus links the ADC to the controller
D_out	8 bit	data out bus. this bus sends outputs the digital data to the main controller
D_out_st	bit	data out strobe. this signals the main controller when a new value is placed on the output bus

B.5.2 Internal Signals

The various state types are explained as comments in the actual VHDL and therefore does not require separate listing.

Appendix C

VHDL Code

C.1 EDAC Controller

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.All;
USE ieee.numeric_std.ALL;

entity Edac_Controller is
port(   clk             : in STD_LOGIC;
        C_reset_enc     : in STD_LOGIC;
        C_reset_dec     : in STD_LOGIC;
        C_Bypass_enc    : in STD_LOGIC;
        C_Bypass_dec    : in STD_LOGIC;
        C_Err_Message   : out STD_LOGIC_VECTOR(2 downto 0);
        C_Err_M_str     : out STD_LOGIC;

        E_Datain        : in STD_LOGIC_VECTOR(7 DOWNTO 0);
        E_Datain_str    : in STD_LOGIC;
        E_Buffer_Rdreq  : in STD_LOGIC;
        E_Buffer_Empty  : out STD_LOGIC;
        E_Dataout       : out STD_LOGIC_VECTOR(7 DOWNTO 0);

        D_Datain        : in STD_LOGIC_VECTOR(7 DOWNTO 0);
        D_Datain_str    : in STD_LOGIC;
        D_Dataout       : out STD_LOGIC_VECTOR(7 DOWNTO 0);
        D_Data_rdy      : out STD_LOGIC;
        D_Data_get      : in  STD_LOGIC
        );


END entity Edac_Controller;

ARCHITECTURE RTL of Edac_Controller is

------------COMPONENTS-------------------------------
COMPONENT Input_Buffer IS
        PORT
        (
        data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        wrreq           : IN STD_LOGIC ;
        rdreq           : IN STD_LOGIC ;
        clock           : IN STD_LOGIC ;
        q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        full            : OUT STD_LOGIC ;
        empty           : OUT STD_LOGIC
        );
END COMPONENT Input_Buffer;

COMPONENT Output_Buffer IS
        PORT
        (
        data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        wrreq           : IN STD_LOGIC ;
        rdreq           : IN STD_LOGIC ;
        clock           : IN STD_LOGIC ;
```

```
        q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        full            : OUT STD_LOGIC ;
        empty           : OUT STD_LOGIC
        );
END COMPONENT Output_Buffer;

COMPONENT RSEncoder IS
        PORT (
        sysclk   : IN STD_LOGIC;
        reset    : IN STD_LOGIC;
        rsin     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        start    : IN STD_LOGIC;
        enable   : IN STD_LOGIC;
        rsout    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END COMPONENT RSEncoder;

COMPONENT RSDecoder IS
        PORT (
        sysclk   : IN STD_LOGIC;
        reset    : IN STD_LOGIC;
        rsin     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        dsin     : IN STD_LOGIC;
        dsout    : IN STD_LOGIC;
        bypass   : IN STD_LOGIC;
        rsout    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        rdyin    : OUT STD_LOGIC;
        outvalid : OUT STD_LOGIC;
        decfail  : OUT STD_LOGIC;
        numerr   : OUT STD_LOGIC_VECTOR (4 DOWNTO 0)
        );
END COMPONENT RSDecoder;

COMPONENT DEC_input_buffer IS
        PORT
        (
        data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        wrreq           : IN STD_LOGIC ;
        rdreq           : IN STD_LOGIC ;
        clock           : IN STD_LOGIC ;
        q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        full            : OUT STD_LOGIC ;
        empty           : OUT STD_LOGIC ;
        usedw           : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
        );
END COMPONENT DEC_input_buffer;
COMPONENT DEC_output_buffer IS
        PORT
        (
        data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        wrreq           : IN STD_LOGIC ;
        rdreq           : IN STD_LOGIC ;
        clock           : IN STD_LOGIC ;
        q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        full            : OUT STD_LOGIC ;
        empty           : OUT STD_LOGIC
        );
END COMPONENT DEC_output_buffer;
-----------------TYPES-----------------------------------

type state_type0 is (reset0,start0,s1,s2,s2a,s3);
type state_type1 is (reset1,wait_rdy,start1);
-----------------SIGNALS---------------------------------
--ENCODER--
signal  state0          : state_type0;
signal  symbol_num      : integer range 0 to 64;
signal  Bufin_full      :  STD_LOGIC;
signal  Bufin_empty     :  STD_LOGIC;
signal  Bufin_rdreq     :  STD_LOGIC;
signal  Bufin_dataout   :  STD_LOGIC_VECTOR(7 DOWNTO 0);
signal  Enc_reset       :  STD_LOGIC;
signal  Enc_enable      :  STD_LOGIC;
signal  Enc_start       :  STD_LOGIC;
signal  Enc_Dataout     :  STD_LOGIC_VECTOR(7 DOWNTO 0);
signal  Bufout_full     :  STD_LOGIC;
signal  Bufout_Wrreq    :  STD_LOGIC;
signal  Watchdog        :  integer range 0 to 2047;     --data arrival timeout value

--DECODER--
```

```
signal  state1          :  state_type1;
signal  Dec_reset       :  STD_LOGIC;
signal  Dec_out         :  STD_LOGIC;
signal  Dec_rdyin       :  STD_LOGIC;
signal  Dec_outvalid    :  STD_LOGIC;
signal  Dec_decfail     :  STD_LOGIC;
signal  Dec_numerr      :  STD_LOGIC_VECTOR(4 downto 0);
signal  Dec_rsout       :  STD_LOGIC_VECTOR(7 downto 0);
signal  Dec_dsin        :  STD_LOGIC;

signal  Dbufin_rdreq    :  STD_LOGIC;
signal  Dbufin_empty    :  STD_LOGIC;
signal  Dbufin_full     :  STD_LOGIC;
signal  Dbufin_q        :  STD_LOGIC_VECTOR(7 DOWNTO 0);

signal  buffer_strobe   :  STD_LOGIC;
signal  Dbufout_str     :  STD_LOGIC;
signal  Dbufout_full    :  STD_LOGIC;

--MULTIPLEXER--
signal  Muxout          :  STD_LOGIC_VECTOR(7 DOWNTO 0);
signal  Muxout_str      :  STD_LOGIC;
signal  Bypass_str      :  STD_LOGIC;
signal  rcnt            :  integer range 0 to 63;       --used to remove codesymbols at decoder
------------------BEGIN-----------------------------------
BEGIN
g1: Input_buffer port map(E_Datain,E_Datain_str,Bufin_rdreq,clk,Bufin_dataout,Bufin_full,
                          Bufin_empty);

g2: Output_buffer port map(Muxout,Muxout_str,E_Buffer_Rdreq,clk,E_Dataout,Bufout_full,
                          E_Buffer_Empty);

g3: RSEncoder port map(clk,Enc_reset,Bufin_dataout,Enc_start,Enc_Enable,Enc_Dataout);

g4: RSDecoder port map(clk,Dec_reset,Dbufin_q,Dec_dsin,Dec_out,C_Bypass_dec,Dec_rsout,
                          Dec_rdyin,Dec_outvalid,Dec_decfail,Dec_numerr);

g5: DEC_input_buffer port map(D_Datain,D_Datain_str,Dbufin_rdreq,clk,Dbufin_q,Dbufin_full,
                          Dbufin_empty,TEST_capacity);

g6: DEC_output_buffer port map(Dec_rsout,buffer_strobe,D_Data_get,clk,D_Dataout,Dbufout_full,
                          D_Data_rdy);

Muxout  <=Enc_dataout when C_Bypass_Enc='0' else        --bypass encoder when set
                Bufin_dataout;
Muxout_str<=Bufout_Wrreq when C_Bypass_Enc='0' else     --bypass encoder strobe
                Bypass_str;
Dec_out <= '1';                                         --output data into the output buffer as soon as
                                                        --it is available
buffer_strobe<=Dec_outvalid when rcnt<43 else
                '0';                                    --causes only the data symbols to be inputted
                                                        --into the output buffer
-------------------------------------------------------------------------------
--This process controls the encoder, encoder input and output buffers.
encode_ctrl: process(clk,C_reset_enc)
begin
        if C_reset_enc= '1' then
                enc_reset       <='1';          --clears contents of encoder
                enc_start       <='0';
                state0<=reset0;
        elsif rising_edge(clk) then
                Bufin_rdreq     <='0';
                enc_enable      <='0';          --controls symbol input to encoder
                ------------------------
                Bufout_Wrreq<=Enc_enable;       --writes encoder value into output buffer
                ------------------------
                Bypass_str      <=Bufin_rdreq;  --D-latch with read str
                CASE state0 is
                        when reset0 =>
                                symbol_num      <=0;
                                enc_start       <='0';
                                Watchdog        <=0;
                                if Bufin_empty='0' then
                                        enc_reset       <='0';
                                        Bufin_rdreq     <='1';
                                        state0          <=start0;
                                end if;

                        when start0 =>
```

```
                enc_start       <='1';
                enc_enable      <='1';
                symbol_num      <=symbol_num+1; --number of symbols in codeword
                state0          <=s1;

        when s1 =>
                enc_start       <='0';
                Watchdog        <=Watchdog+1;
                if watchdog=2047 then           --timout showing end of data stream
                        state0<=s2a;
                elsif symbol_num=43 then        -- 43= total codeword - code symbols
                        state0<=s3;
                elsif Bufin_empty='0' then
                        Bufin_rdreq     <='1';
                        state0<=s2;
                end if;

        when s2 =>
                enc_enable      <='1';
                symbol_num      <=symbol_num+1;
                state0          <=s1;

        -- this state fills up rest of codeword
        when s2a =>
                watchdog        <=0;
                if symbol_num=43 then
                        state0<=s3;
                        enc_enable      <='0';
                else
                        enc_enable      <='1';
                        symbol_num      <=symbol_num+1;
                end if;

        -- this state pulses in the code symbols into the data stream
        when s3 =>
                symbol_num      <=symbol_num+1;
                if symbol_num = 64 then         --codeword complete
                        state0<=reset0;
                else
                        enc_enable      <='1';
                end if;
        end case;

    end if;
end process encode_ctrl;
-------------------------------------------------------------------------------
--This process controls the decoder, decoder input and output buffers.
Decode_ctrl: process(clk,C_reset_dec)
begin
    if C_reset_dec= '1' then
            Dec_reset       <='1';
            Dec_dsin        <='0';
            state1<=reset1;
    elsif rising_edge(clk) then
            Dec_dsin        <='0';
            Dbufin_rdreq<='0';
            CASE state1 is
                when reset1 =>
                    Dec_reset       <='1';
                    State1<=wait_rdy;

                when wait_rdy =>
                    Dec_reset       <='0';
                    if Dbufin_empty='0' and Dec_rdyin='1' then
                                --shows that data has arrived and decoder is ready
                        Dbufin_rdreq<='1';      --gets  symbol from fifo
                        state1          <=start1;
                    end if;

                when start1=>
                    Dec_dsin        <='1'; --puls symbol into decoder
                    state1<=wait_rdy;

            end case;
    end if;
end process Decode_ctrl;
-------------------------------------------------------------------------------
--This process counts the symbols per codeword when outputted from the decoder
remove_codesymbols:process(clk,Dec_outvalid)
```

```
begin
        if Dec_outvalid='0' then        --this signal is high when data is outputted
                rcnt<=0;
        elsif rising_edge(clk) then
                rcnt<=rcnt+1;
        end if;
end process remove_codesymbols;
-------------------------------------------------------------------
--This process reports to the master controller when an error has occurred
error_ctrl : process(clk, bufin_full, bufout_full, dec_decfail, Dbufin_full, Dbufout_full)
begin
        if bufin_full='1' then
                C_Err_Message<="001";
                C_Err_M_str<='1';
        elsif bufout_full='1' then
                C_Err_Message<="010";
                C_Err_M_str<='1';
        elsif Dec_decfail='1' then
                C_Err_Message<="011";
                C_Err_M_str<='1';
        elsif Dbufin_full='1' then
                C_Err_Message<="100";
                C_Err_M_str<='1';
        elsif Dbufout_full='1' then
                C_Err_Message<="101";
                C_Err_M_str<='1';
        else
                C_Err_Message<="000";
                C_Err_M_str<='0';
        end if;
end process error_ctrl;
end architecture rtl;
```

C.2 Master Controller and Bad Block Controller

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.All;
USE ieee.numeric_std.ALL;

entity Main_Cont is
port(   clk             : IN    STD_LOGIC;
        Cur_In          : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        cur_in_str      : IN    STD_LOGIC;

        EDAC_reset_enc  : OUT STD_LOGIC;
        EDAC_reset_dec  : OUT STD_LOGIC;
        EDAC_Bypass_enc : OUT STD_LOGIC;
        EDAC_Bypass_dec : OUT STD_LOGIC;

        F_CLE           : IN    STD_LOGIC;
        F_Data          : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        F_RdyBsy        : IN    STD_LOGIC;
        F_ALE           : IN    STD_LOGIC;
        F_WE            : IN    STD_LOGIC;
        F_RE            : IN    STD_LOGIC;

        F_POWER_CTRL    : OUT   STD_LOGIC;

        Reset_Flash     : OUT   STD_LOGIC;

        WD_En           : OUT   STD_LOGIC;
        WD_Timeout      : IN    STD_LOGIC;

        BBR_Adr         : IN    STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
        BBR_Puls        : IN    STD_LOGIC;
        BBR_Dataout     : OUT   STD_LOGIC_VECTOR (0 DOWNTO 0);
        BBR_Strout      : OUT   STD_LOGIC;
        BBW_Adr         : IN    STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
        BBW_Puls        : IN    STD_LOGIC;
        BBW_Datain      : IN    STD_LOGIC_VECTOR (0 DOWNTO 0);

        EDAC_error_msg  : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
        EDAC_errmsg_str : IN    STD_LOGIC;

        SER_DATA        : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
        SER_STR         : OUT   STD_LOGIC;
```

```
        SER_RX_DATA     : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        SER_RX_RD       : OUT   STD_LOGIC;
        SER_RX_RDY      : IN    STD_LOGIC
        );
END entity Main_Cont;
--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ARCHITECTURE RTL of Main_Cont is

component BB_CONTROLLER is
port
        (
        clk             :in     STD_LOGIC;
        reset           :in     STD_LOGIC;

        R_Adr           :in     STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
        R_Puls          :in     STD_LOGIC;
        R_Dataout       :out    STD_LOGIC_VECTOR (0 DOWNTO 0);
        R_Strout        :out    STD_LOGIC;

        W_Adr           :in     STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
        W_Puls          :in     STD_LOGIC;
        W_Datain        :in     STD_LOGIC_VECTOR (0 DOWNTO 0)
        );
END component BB_CONTROLLER;

constant        BBadrWidth      : integer :=12; --Nr of bits for block address
constant        AdrWidth        : integer :=25; --Nr of bits for flash address
constant        PageWidth       : integer :=5;  --Nr of bits for page address

------------------------------------------------------------------
constant        aan             : std_logic:='0';
constant        af              : std_logic:='1';
-------------------COMMAND HANDLER---------------------------------
constant        read1   : std_logic_vector(2 downto 0):="001";
constant        readID  : std_logic_vector(2 downto 0):="010";
constant        reset   : std_logic_vector(2 downto 0):="011";
constant        pprog   : std_logic_vector(2 downto 0):="100";
constant        cbprog  : std_logic_vector(2 downto 0):="101";
constant        erase   : std_logic_vector(2 downto 0):="110";
constant        status  : std_logic_vector(2 downto 0):="111";
-------------------CURRENT HANDLER---------------------------------------
constant        cut_off         : std_logic_vector(2 downto 0):="001";
constant        standby         : std_logic_vector(2 downto 0):="010";
constant        operating       : std_logic_vector(2 downto 0):="011";
constant        high            : std_logic_vector(2 downto 0):="101";
constant        dangerous       : std_logic_vector(2 downto 0):="110";
-------------------ERROR MESSAGES----------------------------------------
constant        NO_RYBY         : std_logic_vector(1 downto 0):="01";
constant        RYBY_SUSP       : std_logic_vector(1 downto 0):="10";
-------------------MPU ERROR CODES TO OBC--------------------------------
constant        bb_error        : std_logic_vector(3 downto 0):="0001";
constant        SEFI_error      : std_logic_vector(3 downto 0):="0010";
constant        PS_error        : std_logic_vector(3 downto 0):="0011";
constant        EDAC_error      : std_logic_vector(3 downto 0):="0100";
constant        SEL_error       : std_logic_vector(3 downto 0):="0101";
constant        cutoff_pwr      : std_logic_vector(3 downto 0):="0110";
-------------------COMMAND CODES FROM OBC--------------------------------
constant        reset_enc       : std_logic_vector(7 downto 0):="00000001";
constant        byp_enc         : std_logic_vector(7 downto 0):="00000010";
constant        reset_dec       : std_logic_vector(7 downto 0):="00000100";
constant        byp_dec         : std_logic_vector(7 downto 0):="00001000";
constant        reset_BB        : std_logic_vector(7 downto 0):="00010000";
-------------------------------------------------------------------------
type    statetype_WD is (idle,running,wait_high,timeout);
type    statetype_SEFI is(idle,reset1,reset2,reset3,erase1,prog1);
type    statetype_latch is(idle,delay,BIGWAIT,latch1,latch2);
type    statetype_P is(idle,power_on,power_cycle,power_cycle2,power_cycle3,
                                        power_remove,power_remove2);
type    statetype_A is(idle,st1,st2,st3,st4,st5,st6,st7,st8);
type    statetype_BB is(idle,check1,check2,update1,update2,update3);
type    statetype_M is(idle,edac1,power1,bb1,bb2,bb3,bb4,bb5,sefi1,sel1,
                                        pwr1,end_byte);
type    statetype_O is(idle,s1);

-------------------WATCHDOG SIGNALS--------------------------------------
signal  F_ryby          : std_logic;
signal  stateWD         : statetype_WD;
signal  proc_complete   : std_logic;
signal  WD_Error_Message: std_logic_vector(1 downto 0);
```

```
signal  transition      : std_logic;
------------------COMMAND SIGNALS-----------------------------------
signal  CUR_FUNC        : std_logic_vector(2 downto 0);
signal  Start_WD        : std_logic;
------------------CURRENT SIGNALS-----------------------------------
signal  CUR_CURRENT     : std_logic_vector(2 downto 0);
signal  danger_cur      : integer range 0 to 3;
------------------SEFI SIGNALS-----------------------------------
signal  State                   : statetype_SEFI;
signal  send_reset_to_flash     : std_logic;
signal  reset_wdog              : std_logic;
signal  Spower_cycle            : std_logic;
------------------LATCHUP SIGNALS-----------------------------------
signal  stateLatch              : statetype_Latch;
signal  cut_power, pwr_rem      : std_logic;
signal  Lpower_cycle            : std_logic;
------------------POWER SIGNALS-----------------------------------
signal  stateP                  : statetype_P;
signal  ERROR_POWERSWITCH       : std_logic;
signal  F_POWER                 : std_logic;
------------------ADRESS SIGNALS-----------------------------------
signal  stateA                  : statetype_A;
signal  CUR_ADRESS, adress      : std_logic_vector(AdrWidth-1 downto 0);
------------------BADBLOCK SIGNALS-----------------------------------
signal  stateBB                 : statetype_BB;
signal  temp_status             : std_logic_vector(7 downto 0);
------------------BADBLOCK COUNTER SIGNALES-----------------------------------
signal  new_bad_block           : std_logic;
signal  Num_of_Bad_Blocks       : std_logic_vector(15 downto 0);
------------------BAD BLOCK CONNECTION SIGNALS-----------------------------------
signal BB_Data_in               :std_logic_vector(0  downto 0);
signal BB_W_Adr                 :std_logic_vector(BBadrWidth-1 downto 0);
signal BB_Wpuls                 :std_logic;
signal resetBB                  :std_logic;
------------------MPU HANDLER SIGNALS-----------------------------------
signal stateM                   : statetype_M;
------------------OBC COMMAND HANDLER SIGNALS-----------------------------------
signal stateO                   : statetype_O;
BEGIN
--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXBEGINNINGXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                -----------------------------------------------------------------------------------------------------------
g1: BB_CONTROLLER port map(clk, resetBB, BBR_Adr, BBR_Puls, BBR_Dataout,
                        BBR_Strout, BB_W_Adr, BB_Wpuls, BB_Data_in);
F_POWER_CTRL<=F_POWER;
Reset_Flash<=send_reset_to_flash;
------------------COMMAND HANDLER-----------------------------------------------------------
Command_Handler: process(clk, F_CLE, f_data, cur_func)
begin
        if rising_edge(clk) then
                if (F_CLE='1' and F_WE='0') then
                        if (f_data ="00000000" or f_data ="00000001" or f_data ="01010000") then
                                Cur_Func<=      read1;
                                Start_WD<='1';
                        elsif f_data ="10010000" then
                                Cur_Func<= readID;
                                Start_WD<='0';
                        elsif f_data ="11111111" then
                                Cur_Func<= reset;
                                Start_WD<='1';
                        elsif f_data ="10000000" then
                                Cur_Func<= PProg;
                                Start_WD<='1';
                        elsif f_data ="00000011" then
                                Cur_Func<= CBProg;
                                Start_WD<='1';
                        elsif f_data ="01100000" then
                                Cur_Func<= Erase;
                                Start_WD<='1';
                        elsif (f_data ="01110000" or f_data ="01110001") then
                                Cur_Func<= Status;
                                Start_WD<='0';
                        else
                                cur_func<=cur_func;
                                Start_WD<='0';
                        end if;
                else
                        Start_WD<='0';
                end if;
```

```
        end if;
end process Command_Handler;
-------------------CURRENT HANDLER-----------------------------------------------------
Current_Handler: process(clk,cur_in_str)
begin
        if rising_edge(clk) and cur_in_str='1' then
                if Cur_in < "00000001"  then     --1
                        CUR_CURRENT     <=cut_off;
                        danger_cur<=0;
                elsif  Cur_in < "00100001" then --33
                        CUR_CURRENT     <=operating;
                        danger_cur<=0;
                elsif  Cur_in < "01000011" then --67
                        CUR_CURRENT     <=high;
                        danger_cur<=0;
                elsif Cur_in < "11111111"  then  --67
                        CUR_CURRENT     <=dangerous;
                        danger_cur<=danger_cur+1;
                else
                        CUR_CURRENT     <=CUR_CURRENT;
                end if;
        end if;
end process Current_Handler;
-------------------WATCHDOG HANDLER----------------------------------------------------
Watchdog_Handler: process(clk,WD_Timeout,F_Ryby,reset_wdog)
begin
        if reset_wdog='1' then
                stateWD <=idle;
                WD_En   <='0';
                transition<='0';
        elsif rising_edge(clk) then
                F_ryby<=F_RdyBsy;                       --synchronizes ready/busy input signal
                case stateWD is
                        when idle=>
                                proc_complete   <='0'; --just to zero 'complete' pulse
                                WD_En   <='0';                  --timer off;
                                WD_Error_Message<="00"; --reset error message;
                                if Start_WD='1' then
                                        stateWD <=running;
                                        transition<='0';
                                elsif F_Ryby='0' and transition='1' then --for sequential row read
                                        transition<='0';
                                        stateWD <=wait_high;
                                elsif F_Ryby='1' then
                                        transition<='1';
                                end if;
                        when running=>                          --waiting for RyBy to go low
                                WD_En   <='1';                  --timer on;
                                if F_Ryby = '0' then
                                        stateWD <=wait_high;
                                elsif WD_Timeout = '1' then
                                        WD_Error_Message<=NO_RYBY;
                                        stateWD <=timeout;
                                end if;
                        when wait_high=>                        --waiting for RyBy to go high
                                WD_En   <='1';
                                if F_Ryby = '1' then
                                        proc_complete<='1';     --pulse to inform completion of process
                                        stateWD <=idle;
                                elsif WD_Timeout = '1' then
                                        WD_Error_Message<=RYBY_SUSP;
                                        stateWD <=timeout;
                                end if;
                        when timeout=>
                                WD_En   <='0';
                end case;
        end if;
end process Watchdog_handler;
-------------------SEFI HANDLER--------------------------------------------------------
SEFI_Handler:   process(clk,WD_Timeout,F_POWER)
begin
        if rising_edge(clk) then
                if F_POWER=af  then
                        reset_wdog<='1';        --gets wdog out of timeout phase
                        send_reset_to_flash<='1';
                        state<=idle;
                else
                        reset_wdog<='0';
                        send_reset_to_flash<='0';
```

```
                        Spower_cycle<='0';
                        case state is
                                when idle=>     --waits for timeout to occur
                                        if WD_Timeout = '1' then
                                                if Cur_Func=read1               then    state<=reset1;
                                                elsif Cur_Func=erase    then    state<=erase1;
                                                elsif (Cur_Func=pprog or Cur_Func=cbprog) then
                                                        state<=prog1;
                                                elsif Cur_Func=reset    then    state<=reset1;
                                                end if;
                                        end if;

                                when erase1=>           --Check which of the 2 errors occured
                                        if WD_Error_Message=RYBY_SUSP then      --Block_erase SEFI
                                                Spower_cycle<='1';
                                                state<=idle;
                                        else                                    --Partial erase SEFI
                                                state<=reset1;          --must repeat erase process;
                                        end if;

                                when prog1=>
                                        state<=reset1;

                                when reset1=>
                                        reset_wdog<='1';                --set wdog to wait for command
                                        send_reset_to_flash<='1';       --sends reset command
                                        state<=reset2;
                                when reset2 =>
                                        state<=reset3;
                                when reset3 =>
                                        if WD_Timeout='1' then      --wait for outcome of reset command
                                                Spower_cycle<='1';      --either timeout or sucessful
                                                reset_wdog<='1';
                                                state<=idle;
                                        elsif proc_complete='1' then --if timeout then reset power
                                                state<=idle;
                                        end if;

                        end case;
                end if;
        end if;
        end process SEFI_handler;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Latchup_Handler: process(clk,proc_complete,cur_current)
        VARIABLE BIGW : INTEGER RANGE 0 TO 25000000;
begin
        if rising_edge(clk) then
                cut_power<='0';
                Lpower_cycle<='0';
                case stateLatch is
                        when idle=>
                                BIGW:=0;
                                pwr_rem<='0';
                                if cur_current=dangerous and danger_cur=2 then  --checks for high
                                     current
                                        Lpower_cycle<='1';
                                        stateLatch<=delay;              --give time to switch off!
                                end if;
                        when delay=>
                                if cur_current=cut_off then
                                        stateLatch<=BIGWAIT;
                                end if;
                        WHEN BIGWAIT=>
                                BIGW:=BIGW+1;                   --NEED TO PUT IN DELAY FOR SWITCHING
                                     PURPOSES
                                IF BIGW=0 THEN                  --gives time to turn pot!!!
                                        STATELATCH<=LATCH1;
                                END IF;

                        when latch1=>
        --if current turned back on and still dangerous then power removed
                                if cur_current=dangerous then
                                        stateLatch<=latch2;
                                        pwr_rem<='1';
        --if procedure completes then chip has recovered
                                elsif proc_complete='1' then
                                        stateLatch<=idle;
                                end if;
```

```
                        when latch2=>
                                cut_power<='1';
                                pwr_rem<='0';
                end case;
        end if;
end process latchup_handler;
-------------------------------------------------------------------------------
POWER: process(clk)
variable power_cnt      : integer range 0 to 33554431;
variable cut_time       : integer range 0 to 255;
begin
        if rising_edge(clk) then
                case stateP is
                        when idle=>             --allows signals to reset
                                stateP<=power_on;
                        when power_on=>
                                ERROR_POWERSWITCH<='0';
                                F_power<=aan;
                                power_cnt       :=0;
                                cut_time        :=0;
                                if cut_power='1' then
                                        stateP<=power_remove;
                                elsif Lpower_cycle='1' or Spower_cycle='1' then
                                        stateP<=power_cycle;
                                end if;

                        when power_cycle=>
                                F_POWER<=af;
                                cut_time:=cut_time+1;
                                if cur_current=cut_off then     --check that power is cut off!
                                        cut_time:=0;
                                        stateP<=power_cycle2;
                                elsif cut_time=0 then
                                        ERROR_POWERSWITCH<='1';
                                        stateP<=idle;
                                END IF;
                        when power_cycle2=>             --remove power for set time
                                power_cnt:=power_cnt+1;
                                if power_cnt=0 then
                                        stateP<=power_cycle3;
                                end if;
                        when power_cycle3=>             --check power is restored
                                cut_time:=cut_time+1;
                                F_power<=aan;
                                if (cur_current>cut_off) then   --check that power is restored!
                                        stateP<=power_on;
                                elsif cut_time=0 then
                                        ERROR_POWERSWITCH<='1';
                                        stateP<=idle;
                                END IF;

                        when power_remove=>
                                F_POWER<=af;
                                cut_time:=cut_time+1;
                                if cur_current=cut_off then     --check that power is cut off!
                                        stateP<=power_remove2;
                                elsif cut_time=0 then
                                        ERROR_POWERSWITCH<='1';
                                        stateP<=idle;
                                END IF;
                        when power_remove2=>
                end case;
        end if;
end process power;
-------------------------------------------------------------------------------
Adress_handler : process(clk,F_ALE,F_WE,adress)
begin
        if F_ALE='0' then
                stateA<=idle;
                CUR_ADRESS<=adress;
        elsif rising_edge(clk) then
                case stateA is
                        when idle=>
                                if F_WE='0' then
                                        if Cur_Func=Erase then
                                                stateA<=st3;
                                        else
                                                stateA<=st1;
                                        end if;
```

```
                        end if;
                when st1=>
                        if F_WE='1' then
                                adress(7 downto 0)<=F_Data;
                                stateA<=st2;
                        end if;
                when st2=>
                        if F_WE='0' then
                                stateA<=st3;
                        end if;
                when st3=>
                        if F_WE='1' then
                                adress(15 downto 8)<=F_Data;
                                stateA<=st4;
                        end if;
                when st4=>
                        if F_WE='0' then
                                stateA<=st5;
                        end if;
                when st5=>
                        if F_WE='1' then
                                adress(23 downto 16)<=F_Data;
                                stateA<=st6;
                        end if;
                when st6=>
                        if F_WE='0' then
                                stateA<=st7;
                        end if;
                when st7=>
                        if F_WE='1' then
                                adress(AdrWidth-1 downto 24)<=F_Data((AdrWidth-25)downto 0);
                                stateA<=st8;
                        end if;
                when st8=>
                        CUR_ADRESS<=adress;      --lastly set current address to collected value
        end case;
    end if;
end process adress_handler;
-------------------------------------------------------------------------------
BB_Table_Handler : process(clk,Cur_Func,F_RE)
begin
    if rising_edge(clk) then
        case stateBB is
            when idle=>
                BB_Wpuls  <='0';                --ensure zero write pulse
                BB_Data_in<="0";
                temp_status<=(others=>'0');
                if Cur_func=status and F_RE='0' then --check status byte
                    stateBB<=check1;
                elsif BBW_Puls='1' then         --External Update
                    BB_W_Adr<=BBW_Adr;
                    BB_Data_in<=BBW_Datain;
                    stateBB<=update2;
                end if;

            when check1=>
                if F_RE='1' then                --read in status byte
                    temp_status<=F_Data;
                    stateBB<=check2;
                end if;
            when check2=>                       --check LSB for a '0'=pass or '1'=fail
                if temp_status(0)='0' then
                    stateBB<=idle;              --op sucessful
                else
                    stateBB<=update1;           --op unsuccessful
                end if;
            -- update Bad Block table
            when update1 =>
                BB_W_Adr<=CUR_ADRESS(AdrWidth-1 downto (AdrWidth-BBadrWidth)); --chops
                      of 32 page count
                BB_Data_in<="1";                        --sets flag
                stateBB<=update2;
            when update2 =>
                BB_Wpuls  <='1';                        --pulses write
                stateBB<=update3;
            when update3 =>
                BB_Wpuls  <='0';
                stateBB<=idle;
        end case;
```

```
        end if;
end process     BB_Table_Handler;

Bad_Block_Counter : process(clk,ResetBB,BBW_Puls,BBW_Datain)
begin
        if ResetBB='1' then
                Num_of_Bad_Blocks<=(others=>'0');
        elsif rising_edge(clk) then
                if (BB_Wpuls = '1' and  BB_Data_in(0) = '1') then
                        Num_of_Bad_Blocks<=Num_of_Bad_Blocks+1;
                        new_bad_block<='1';             --signals report to be sent to OBC
                else
                        new_bad_block<='0';
                end if;

        end if;
end process Bad_Block_Counter;

MPU_error_handler : process(clk,EDAC_errmsg_str,ERROR_POWERSWITCH)
begin
        if rising_edge(clk) then
                SER_STR<='0';
                case stateM is
                        when idle=>
                                if EDAC_errmsg_str='1' then
                                        stateM<=edac1;
                                elsif Error_Powerswitch='1' then
                                        stateM<=power1;
                                elsif new_bad_block='1' then
                                        stateM<=bb1;
                                elsif WD_Timeout = '1' then
                                        stateM<=sefi1;
                                elsif Lpower_cycle='1' then
                                        stateM<=sel1;
                                elsif pwr_rem='1' then
                                        stateM<=pwr1;
                                end if;
                        ------------------------------------------------
                        ------------------------------------------------
                        when edac1=>
                                SER_DATA<=EDAC_error & "0" & EDAC_error_msg;    --send error message
                                SER_STR<='1';
                                stateM<=end_byte;
                        ------------------------------------------------
                        when power1=>
                                SER_DATA<=PS_error & "0000";
                                SER_STR<='1';
                                stateM<=end_byte;
                        ------------------------------------------------
                        when bb1=>
                                SER_DATA<=bb_error & "0000";
                                SER_STR<='1';
                                stateM<=bb2;
                        when bb2=>
                                SER_DATA<=CUR_ADRESS(20 downto 13);
                                SER_STR<='1';
                                stateM<=bb3;
                        when bb3=>
                                SER_DATA<="0000" & CUR_ADRESS(24 downto 21);
                                SER_STR<='1';
                                stateM<=bb4;
                        when bb4=>
                                SER_DATA<=Num_of_Bad_Blocks(7 downto 0);
                                SER_STR<='1';
                                stateM<=bb5;
                        when bb5=>
                                SER_DATA<=Num_of_Bad_Blocks(15 downto 8);
                                SER_STR<='1';
                                stateM<=end_byte;
                        ------------------------------------------------
                        when sefi1=>
                                SER_DATA<=SEFI_error    & '0' & Cur_Func;
                                SER_STR<='1';
                                stateM<=end_byte;
                        ------------------------------------------------
                        when sel1=>
                                SER_DATA<=SEL_error & "0000";
                                SER_STR<='1';
                                stateM<=end_byte;
```

```
                        ------------------------------------------------
                        when pwr1=>
                                SER_DATA<=cutoff_pwr & "0000";
                                SER_STR<='1';
                                stateM<=end_byte;
                        ------------------------------------------------
                        when end_byte=>
                                SER_DATA<=(others=>'0');
                                        --send end byte
                                SER_STR<='1';
                                stateM<=idle;
                        ------------------------------------------------
                end case;
        end if;
end process MPU_error_handler;

OBC_Command_Handler :process(clk,SER_RX_RDY)
begin
        if rising_edge(clk) then
                SER_RX_RD<='0';
                EDAC_reset_enc<='0';
                EDAC_reset_dec<='0';
                resetBB<='0';
                case stateO is
                        when idle=>
                                if SER_RX_RDY='0' then
                                        SER_RX_RD<='1';
                                        stateO<=s1;
                                end if;
                        when s1=>
                                case SER_RX_DATA is
                                        when reset_enc=> EDAC_reset_enc<='1';
                                                         EDAC_Bypass_enc<='0';
                                        when reset_dec=> EDAC_reset_dec<='1';
                                                         EDAC_Bypass_dec<='0';
                                        when byp_enc=>   EDAC_Bypass_enc<='1';
                                        when byp_dec=>   EDAC_Bypass_dec<='1';
                                        when reset_bb=>  resetBB<='1';
                                        when others=>

                                end case;
                                stateO<=idle;
                end case;
        end if;
end process OBC_Command_Handler;

END ARCHITECTURE RTL;


-------------------------------------------------
--Bevan Bryer   04/02/2004
--New Bad Block Contoller Program to handle
--writing and reading from the dual port ram
--on the cyclone chip
--ver 1.0
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.ALL;

entity BB_CONTROLLER is
port
        (
        clk             :       in      STD_LOGIC;
        reset           :       in      STD_LOGIC;

        R_Adr           :       in      STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
        R_Puls          :       in      STD_LOGIC;
        R_Dataout       :       out     STD_LOGIC_VECTOR (0 DOWNTO 0);
        R_Strout        :       out     STD_LOGIC;

        W_Adr           :       in      STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
        W_Puls          :       in      STD_LOGIC;
        W_Datain        :       in      STD_LOGIC_VECTOR (0 DOWNTO 0)
        );
END entity BB_CONTROLLER;

ARCHITECTURE RTL OF BB_CONTROLLER is
--RAMBLOCK WITHIN FPGA TO STORE BB TABLE
constant        BBadrWidth      : integer :=12; --Nr of bits for block address
```

```
COMPONENT RAM2047x1 IS
        PORT
        (
                data_a          : IN STD_LOGIC_VECTOR (0 DOWNTO 0);
                wren_a          : IN STD_LOGIC  := '1';
                address_a       : IN STD_LOGIC_VECTOR (BBadrWidth-1 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (0 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (BBadrWidth-1 DOWNTO 0);
                wren_b          : IN STD_LOGIC  := '1';
                clock_a         : IN STD_LOGIC ;
                enable_a        : IN STD_LOGIC  := '1';
                clock_b         : IN STD_LOGIC ;
                enable_b        : IN STD_LOGIC  := '1';
                q_a             : OUT STD_LOGIC_VECTOR (0 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (0 DOWNTO 0)
        );
END COMPONENT RAM2047x1;

type Rstate_type is (idle,state1,state2,state3);
type Wstate_type is (resetW,idle,state1);

signal Rstate: Rstate_type;
signal Wstate: Wstate_type;

signal  A_Adr           :       STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
signal  A_Dataout       :       STD_LOGIC_VECTOR (0 DOWNTO 0);
signal  A_Wen           :       STD_LOGIC;
signal  A_Clk_en        :       STD_LOGIC;
signal  A_Datain        :       STD_LOGIC_VECTOR (0 DOWNTO 0);

signal  B_Adr           :       STD_LOGIC_VECTOR(BBadrWidth-1 downto 0);
signal  B_Dataout       :       STD_LOGIC_VECTOR (0 DOWNTO 0);
signal  B_Wen           :       STD_LOGIC;
signal  B_Clk_en        :       STD_LOGIC;
signal  B_Datain        :       STD_LOGIC_VECTOR (0 DOWNTO 0);

BEGIN

g1: RAM4096x1 port map( A_Dataout,A_Wen,A_Adr,B_dataout,B_Adr,
                        B_Wen,clk,A_Clk_en,clk,B_Clk_en,A_Datain,
                        B_Datain);

-------------------------------------------------------------
--process that handles read action
-------------------------------------------------------------
Read:   process (clk,reset) is
                begin
                if reset='1' then
                        Rstate<= idle;
                        A_Clk_en<='0';  --make sure during reset nothing happens
                        A_Dataout<="1"; --just to stop warnings
                elsif rising_edge(clk) then
                        A_Clk_en<='0';
                        A_Wen   <='0';
                        R_Strout<='0';

                        CASE Rstate is

                                when
                                        idle =>
                                                A_Dataout<="0";
                                                if R_puls='1' then
                                                        A_Adr<=R_Adr;
                                                        A_Clk_en<='1';
                                                        Rstate<=state1;
                                                end if;
                                when
                                        state1=>
                                                A_Clk_en<='1';
                                                Rstate<=state2;
                                when
                                        state2=>
                                                Rstate<=state3;
                                when
                                        state3=>

                                                R_Dataout<=A_Datain;
                                                R_Strout<='1';
                                                Rstate<=idle;
```

```
                        end CASE;
                end if;
        end process Read;

-------------------------------------------------------------
--process that handles write action
-------------------------------------------------------------
Write:  process (clk,reset) is
                begin
                if reset='1' then
                        Wstate<= resetW;
                        B_Clk_en <='0'; --make sure nothing happens in reset
                elsif rising_edge(clk) then
                        B_Clk_en <='0';
                        B_Wen    <='1';

                        CASE Wstate is
                                when
                                        resetW =>
                                                Wstate<=idle;
                                when
                                        idle =>
                                                if W_puls='1' then
                                                        B_Adr<=W_Adr;
                                                        B_Dataout<=W_Datain;
                                                        B_Clk_en <='1';
                                                        Wstate<=state1;
                                                end if;
                                when
                                        state1=>
                                                B_Clk_en <='1';          --lets 1 clock pulse through
                                                Wstate<=idle;
                        end CASE;
                end if;
        end process Write;
-------------------------------------------------------------
end architecture RTL;
```

C.3 Communications

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity baudgen is
        port(   clk     :IN STD_LOGIC;
                puls,
                puls2   :OUT STD_LOGIC);
end baudgen;

architecture gen of baudgen is
type state_type is (counting, high,double);
signal state: state_type;
begin
        process (clk) is
                variable cnt : integer range 0 to 869; --counter to set BAUD rate
                variable cnt2: integer range 0 to 16;
                begin
                        if rising_edge(clk) then
                                CASE state is
                                        when counting =>
                                                puls <= '0';
                                                puls2 <='0';
                                                cnt := cnt + 1;
                                                if (cnt = 868) then
                                                        cnt:=0;
                                                        state <= high;
                                                end if;
                                        when high =>
                                                if cnt2=15 then
                                                        state <= double;
                                                else puls <= '1';
                                                         cnt2 := cnt2+1;
                                                         state <= counting;
                                                end if;
                                        when double =>
                                                puls2 <= '1'; cnt2:=0;
                                                puls  <= '1';
```

```
                                        state <= counting;
                        end CASE;
                end if;
        end process;
end architecture gen;


--this program inputs 8 bit binary and outputs 2 8 bit
--hex values in ascii, LSB's first, then MSB'd

entity bintohex_conv is
        port(   clk             : IN STD_LOGIC;
                reset           : IN STD_LOGIC;
                Datain          : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                not_empty       : IN STD_LOGIC;
                get_byte        : OUT STD_LOGIC;
                Dataout         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                strout          : OUT STD_LOGIC);
end entity bintohex_conv;

architecture rtl of bintohex_conv is
        TYPE state_type is (idle,getbyte,getbyte2,send1,send2,send3,send4,send5,send6);
        signal state    : state_type;
begin
        process(clk,reset,not_empty) is
                variable temp1,temp2    :std_logic_vector(7 downto 0);
                variable int1,int2      :integer range 0 to 15;
                variable cnt    :integer range 0 to 16383;
        begin
        if reset='1' then
                state<=idle;
        elsif rising_edge(clk) then
                strout<='0';
                case state is
                        when idle =>
                                cnt:=1;
                                if not_empty='0' then   --wait for available byte from input buffer
                                        get_byte<='1';
                                        state<=getbyte;
                                else
                                        get_byte<='0';
                                end if;

                        when getbyte=>
                                get_byte<='0';
                                state<=getbyte2;

                        when getbyte2=> --seperates and converts byte
                                int1:=conv_integer(unsigned(Datain(3 downto 0)));
                                int2:=conv_integer(unsigned(Datain(7 downto 4)));
                                state<=send1;

                        when send1 =>   --converts binary to ASCII
                                        if int1<10 then
                                                temp1:=conv_std_logic_vector(int1+48,8);
                                        else
                                                temp1:=conv_std_logic_vector(int1+55,8);
                                        end if;

                                        if int2<10 then
                                                temp2:=conv_std_logic_vector(int2+48,8);
                                        else
                                                temp2:=conv_std_logic_vector(int2+55,8);
                                        end if;
                                        dataout<=temp2;
                                        state<=send2;
                        when
                                send2=> --send first byte
                                        strout<='1';
                                        state<=send3;
                        when
                                send3=>
                                        if cnt=9000 then
                                                state<=send4;
                                        end if;
                                        cnt:=cnt+1;
                        when
                                send4=>
                                        dataout<=temp1;
```

```
                                        state<=send5;
                                        cnt:=1;
                        when
                                send5=> --send second byte
                                        strout <='1';
                                        state<=send6;
                        when
                                send6=>
                                        if cnt=9000 then
                                                state<=idle;
                                        end if;
                                        cnt:=cnt+1;
                end case;
        end if;
end process;
end architecture rtl;

entity uart is
        port(   clk             : IN STD_LOGIC;
                BaudPuls        : IN STD_LOGIC;
                Data            : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                DataStrobe      : IN STD_LOGIC;
                SERIN           : IN STD_LOGIC;
                Data_out        : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                Data_out_str    : OUT STD_LOGIC;
                SERUIT          : OUT STD_LOGIC);
end uart;

architecture trans of uart is
TYPE state_type_T is (idle,xmit);
TYPE state_type_R is (idle,rec,send);

signal stateT           : state_type_T;
signal stateR           : state_type_R;
signal tempStrobe       : STD_LOGIC;
signal txData           : STD_LOGIC_VECTOR(9 DOWNTO 0); -- transmit stream
signal rxData           : STD_LOGIC_VECTOR(7 DOWNTO 0); -- received stream
signal rxCnt            : integer range 0 to 7;

begin
txData <= '1' &  Data & '0'; --adds start and stop bit, and data to create serial packet

transmit: process (clk) is
        variable temp : integer range 0 to 9;
        variable chk : STD_LOGIC;
        begin
                if rising_edge(clk) then
                        case stateT is
                                when idle =>
                                        Seruit <= '1';
                                        if ((tempStrobe = '0') AND (DataStrobe = '1')) then
                                                stateT <= xmit;
                                                chk:='0';
                                                temp := 0;
                                        else
                                                tempStrobe <= DataStrobe;
                                        end if;
                                when xmit =>
                                        if BaudPuls ='1' and chk = '0'  then
                                                Seruit <= txData(temp);
                                                chk:='1';
                                                if (temp = 9) then
                                                        stateT <= idle;
                                                else
                                                        temp := temp + 1;
                                                end if;
                                        elsif Baudpuls ='0' then
                                                chk := '0';
                                        end if;
                        end case;
                end if;
        end process transmit;

receive: process (clk) is
        variable chk :STD_LOGIC;
        begin
                if rising_edge(clk) then
                        case stateR is
                                when idle =>
```

```
                                        Data_out_str <='0';
                                        Data_out<=(others=>'0');
                                        if SERIN='0' and BaudPuls ='1' then
                                                stateR <= rec;
                                                chk:='1';       --set it to '1' so that next state wont
                                                rxCnt <= 0;     --read in start bit
                                        else
                                                chk:='0';
                                                stateR <= idle;
                                        end if;
                                when rec =>
                                        --note: baudpuls could still be 1 when coming from idle state
                                        if BaudPuls ='1' and chk = '0' then
                                                rxData(rxCnt)<=Serin;
                                                chk:='1';
                                                if (rxCnt = 7) then
                                                        stateR <= send;
                                                else
                                                        rxCnt <= rxCnt + 1;
                                                end if;
                                        elsif Baudpuls ='0' then
                                                chk :='0';
                                        end if;
                                when send=>
                                        Data_out_str <='1';
                                        Data_out<=rxData;
                                        stateR <= idle;
                        end case;
                end if;
        end process receive;
end architecture trans;
```

C.4 Watchdog Timer

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.All;
USE ieee.numeric_std.ALL;

entity Watchdog_Timer is
port(   clk             : IN    std_logic;
        Wen             : IN    std_logic;
        Timeout         : OUT   std_logic
        );


END entity Watchdog_Timer;

ARCHITECTURE RTL of Watchdog_Timer is
        type state_type is(counting,timeup,wait_reset);
        signal state    : state_type;
BEGIN
        process (clk,Wen)
        variable        counter         : integer range 0 to 67108863; --25 bit counter;
        begin
                if Wen='0' then
                        state   <=counting;
                        counter :=0;
                        Timeout <='0';
                elsif rising_edge(clk) then
                        case state is
                                when counting =>
                                        timeout <='0';
                                        counter :=counter +1;
                                        if counter = 0 then
                                                state<=timeup;
                                        end if;
                                when timeup     =>
                                        Timeout <='1';
                                        state<=wait_reset;
                                when wait_reset=>
                                        Timeout <='0';
                        end case;
                end if;
        end process;
END ARCHITECTURE RTL;
```

C.5 Analog to digital converter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.All;
USE ieee.numeric_std.ALL;

entity ADC_Cont is
port(   clk,
        reset    : in    std_logic;
        nConvst  : out   std_logic;
        nCSRD    : out   std_logic;
        busy     : in    std_logic;
        D_in     : in    std_logic_vector(7 downto 0);
        D_out    : out   std_logic_vector(7 downto 0);
        D_out_st: out   std_logic
        );
END entity ADC_Cont;

ARCHITECTURE RTL of ADC_Cont is
type state_type is (startup,startup2,startup3,s1,s2,s3,s3a,s4,s5,s6,s7);
signal state                        : state_type;
signal busy_temp                    : std_logic;

BEGIN

process(clk,reset)
        variable delay  : integer range 0 to 63;
        variable delay2 : integer range 0 to 18000;
        begin
                if reset='1' then
                        state<=startup;
                elsif rising_edge(clk) then
                        busy_temp<=busy;                        --synchronize busy signal
                        CASE state is
                                when startup =>
                                        nCSRD<='1';
                                        nConvst<='0';
                                        D_out_st<='0';
                                        state<=startup2;
                                when startup2 =>                --delay so chip power can settle
                                        delay:=delay+1;
                                        if delay=0 then
                                                state<=startup3;
                                        end if;
                                when startup3   =>
                                        nConvst<='1';           --low to high transition
                                        delay:=delay+1;         --wait 1us (50 pulses)
                                        if delay=0 then
                                                state<=s1;
                                        end if;
                                when s1 =>
                                        nConvst<='0';           --high to low transition (2 pulses)
                                        state<=s2;
                                when s2 =>
                                        nConvst<='0';
                                        state<=s3;
                                when s3 =>
                                        nConvst<='1';
                                        if busy_temp='1' then   --make sure busy is high or goes high
                                                state<=s3a;
                                        end if;
                                when s3a=>
                                        if busy_temp='0' then   --wait for busy to go low
                                                state<=s4;
                                        end if;
                                when s4 =>
                                        nCSRD<='0';
                                        state<=s5;
                                when s5 =>
                                        D_out<=d_in & '0';      --sample data and pulse it out
                                        state<=s6;
                                when s6 =>
                                        D_out_st<='1';          --strobe data out
                                        state<=s7;
                                when s7=>
                                        D_out_st<='0';
```

```
                                        nCSRD<='1';
                                        delay2:=delay2+1;
                                        if delay2=18030 then     --18030
                                                delay2:=0;
                                                state<=s1;
                                        end if;

                                end case;
                        end if;
                end process;
end architecture rtl;
```